

3 Distributed Programming

SCCs exist in a distributed system environment, either providing service to (or using services from) other machines in the distributed system. Chapter 2 describes the general nature of a distributed system, including the client and server roles that the SCC might be expected to play during its operation. In this chapter we will discuss various aspects of programming SCCs in a distributed system.

The client-server technology is the dominant communication paradigm used in contemporary distributed systems. In this paradigm the client application interacts with a peer server application using OSI protocol layers at the transport and higher layers. Typically, ST SCC applications use either TCP or UDP transport layer protocols (in conjunction with underlying IP support for internets). This enables the SCC to exploit relatively general types of service, including those to store files and databases, to print information, to host CPU-intensive computations, and generally to make resources available to the SCC applications.

Programming is based on the sequential computation model. Programmers learn about this model before they are able to write even their first program. We will begin the discussion by considering important aspects of sequential computations, thereby providing a foundation for discussing how the work performed by a sequential computation can be distributed across multiple computers to be executed as a set of cooperating sequential computations that behaves as a “distributed program.”

3.1 Sequential and Distributed Computations

Sequential computations are the basis of traditional programming, providing the general rules for how instructions are executed to accomplish an information processing task. Distributed computations are a generalization of sequential computation that defines the pattern in which a *collection* of executing programs work together to solve a common problem.

3.1.1 Sequential Computations

An individual computer is designed to execute a *sequence* of instructions, one after the other. As we know, a computer can be made to do useful work by specifying a sequence of instruction that accomplishes some information processing task, such as sorting a list of integers, creating an invoice, controlling the amount of background noise on an audio channel, guiding a jet airplane, and so on.

The instruction sequence that accomplishes the task is called a **sequential computation** – “sequential” because instructions are executed one after the other (but never at the same time), and “computation” because the collection of instruction executions control the behavior of the computer to accomplish a desired information processing task. Both the nature of the instructions and their order of execution are important in determining the result produced executing a sequential computation.

In your introductory programming class, you learned to design sequential computations in terms of a specific programming language (like C/C++, Java, or some other language). One of the first things you had to learn about programming was that the program’s instructions are executed sequentially beginning at an entry point for an executable code module, and continuing until an instruction causes execution to cease, for example the `exit()` call in C/C++ ceases program execution.

Program execution depends on the **sequential computation rule**, namely that the second instruction in the sequence does not start executing until the first one has completed its execution, the third does not start until the second has finished, and so on.

For example, consider this C code fragment:

```
int x=2;
...
scanf("%d", &x);
```

```
printf("f(%d) = %d\n", x, x*x);
...
```

Lets suppose that, when this code fragment is executed, the user typed “5” causing the Linux-style OS `scanf()` statement to set the value of `x` to 5 when it executes. If the `printf()` were to begin executing before the `scanf()` function completed, then the code fragment might print “f(2) = 4” or “f(5) = 25” (the first output version would occur if the `printf()` used the original value of `x` before the `scanf()` statement was able to store the value 4 into `x`). However according to the sequential computation rule the code fragment can only print “f(5) = 25” since the `scanf()` statement must finish before the `printf()` statement begins. Sequential computation is at the foundation of programming.

An **algorithm** is a strategy for organizing the sequential computation. For example, an exchange sort algorithm will rearrange the numbers in an array so that the smallest (or largest) number is in the first element of the array, the next smallest number is in the second element of the array, and so on. The idea behind the algorithm is that numbers are sorted into ascending order by repeatedly exchanging adjacent numbers if the one in a lower-numbered cell is larger than the one in the next higher numbered cell. Programming rests on the study of algorithms: you have undoubtedly learned a spectrum of algorithms to do linear interpolation, to create and manage a list, to print the numbers in a sorted binary tree, and so on.

A programmer creates a strategy for solving an information processing task by first defining an algorithm (see Figure 3-1). Next, the algorithm is encoded in a particular high level programming language as a sequence of programming language instructions. The resulting source program can then be translated into machine language (for example, so it the program can be executed in a computer with an Intel Pentium processor). After the translation procedure has been completed, the algorithm is represented by a collection of machine instructions that will be executed in a predefined sequence as defined by the control flow instructions (such as loop instructions) within the program. The binary program is stored in a file until it is ready to be executed.

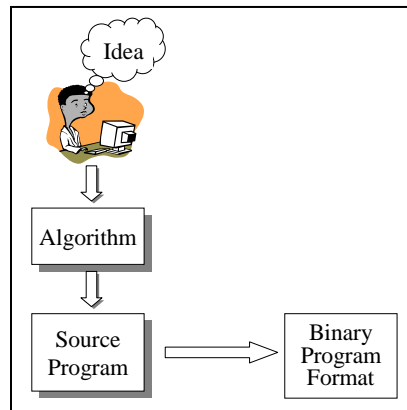


Figure 3-1: Algorithms and Programs

When the user decides to run the program, s/he instructs the operating system to load the program from the designated file into the executable memory. For example in a Unix shell, the user types the name of a file which causes the shell program to load the contents of the specified file into executable memory and to branch to the program’s main entry point. The computer then executes the machine instructions in the program as a sequential computation.

Notice that *the rules of sequential computation do not distinguish between I/O instructions and other instructions*: therefore before the processor can begin to execute the next instruction in the program that follows the input or output instruction, it must wait for the I/O instruction to finish – even if the processor is actively checking the status of the relevant device while it waits for the device to complete the current instruction. In the discussion of Section 2.2, sequential computations should use the schema shown in Figure 2-5(a) in order to assure the sequential computation rule applies to I/O instructions. If the

computation follows the code schema shown in Figure 2-5(b) the system cannot ensure that the sequential computation rule will be true (although the programmer may be able to ensure that it is true because of other aspects of the computation).

We will finish this subsection with an example of a sequential computation that we will modify in various ways to demonstrate various concepts. Consider the sequential computation resulting from executing the program shown in Figure 3-2 (see the supplementary material for a compilable copy of the program). When this program is executed we could expect that this program would copy the contents of a file named “MyInput” to another file named “MyOutput”. The `open()`, `read()`, and `write()` functions are POSIX system calls, but could be ordinary library functions on another OS. The conventional semantics for function calls is that when the calling program calls the function, the calling program suspends until the called program completes its execution and returns. That is, the function executes as if it were a single C instructions. For example, when the `open("MyInput", O_RDONLY)` function is called, the program suspends its own execution and does not resume execution until the `open()` function has returned. In this case, the result from the `open()` call is assigned to the integer variable `inFid`, is set to the value of “-1” if the `open()` failed. Since the `if`-statement cannot be evaluated until `open()` has returned, the computer cannot know whether to next issue an error message (presuming that `inFid` will be nonzero) or to begin opening `MyOutput` (presuming that `inFid` will be zero) until the `open()` function returns a value.

```

#define BUF_LEN 100

int main() {
    int inFid, outFid;
    int numRead;
    int i;
    char buf[BUF_LEN];

    if((inFid = open("MyInput", O_RDONLY)) == -1) {
        fprintf(stderr, "usage: open() failed, halting\n");
        exit(1);
    }
    if((outFid = open("MyOutput", O_CREAT | O_WRONLY)) == -1) {
        fprintf(stderr, "usage: open() failed, halting\n");
        exit(1);
    }

    /* Input & output files are ready to use */
    while((numRead = read(inFid, buf, BUF_LEN)) != 0)
        write(outFid, buf, numRead);    // Copy to MyOutput
    /* File copy complete */

    close(inFid);
    close(outFid);
    exit(0);
}

```

Figure 3-2: Example Sequential Computation

The sequential computation code could be rewritten so that it uses nonblocking I/O operations¹ and so that it controls execution so that the resulting overlap among the processor, input file device, and output file

¹ For example, see the discussion in Chapter 12 of [Stevens, 1992].

device are all overlapped. The code to accomplish this is not necessarily pretty – consult the supplementary materials you are using in conjunction with this booklet. Even more seriously, without using OS synchronization tools that you have not yet learned about, some of these examples could suffer from subtle, infrequently occurring errors that prevent it from being determinate; since these errors are unlikely to occur, we will ignore them until we learn about synchronization.

3.1.2 Autonomous Units of Computation (AUCs)

In Chapter 1, you were introduced to the idea of multiprogramming, in which a single computer can rapidly multiplex execution among a set of sequential computation, executing each for a short period of time. That is, in multiprogramming operating systems, the processor executes each of the computations for a few milliseconds (ms) at a time, so during any second, perhaps a thousand sequential computations could appear to be executing simultaneously, but at a relatively slow rate compared to the raw processor speed. In a multiprogrammed operating systems context, we say that sequential computations are executed **concurrently** if they are actually executed serially yet they *appear* to be executing simultaneously.

Let's elaborate on the discussion in [Section 1.4](#) (also see Figure 1-11): in order to implement multiprogramming, operating systems incorporate code to *simulate a physical machine*, then each sequential computation is allocated its own simulated, abstract (or virtual) machine on which it can execute. By creating abstract machines, the OS can then (potentially) monitor execution and control the way that it allocates hardware resources to each of the abstract machines – and hence to a sequential computation.

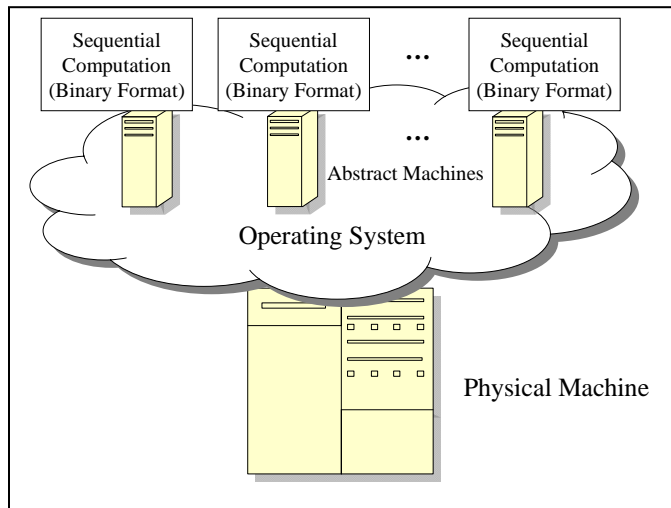


Figure 3-3: Using An Abstract Machine for Each Sequential Computation

Different operating systems use (sometimes radically) different types of abstract machines to implement the concurrency model. Further, different operating systems refer to their abstract machine using a broad spectrum of terminology, ranging across threads, processes, lightweight processes, heavyweight processes, tasks, and so on. Because of this diversity in approaches and terminology, in this book we will refer to a generic abstract machine as an **autonomous unit of computation (AUC)**. Windows NT/XP/Vista class operating systems define an AUC to be a *Windows thread* executing in a *Windows process*. Classic Unix systems (prior to 1995) define an AUC to be a *process*. For the ST class SCC operating systems, we will adapt the classic idea of a *coroutine* to describe an AUC that is used in ST machines. The AUC for an MT class OS is a *thread* – sometimes called a lightweight process.

Now that we have the term AUC, we can talk about an AUC wherever we previously talked about “an executing program” or often when we are thinking about the execution of a sequential computation. If this sounds “AUCward,” © you can informally think of an AUC as a process or a thread, even though we will ultimately provide precise descriptions for each kind of AUC in the context of a class of operating system.

3.1.3 Distributed Computations

A **distributed computation** is a set of cooperating sequential computations.

A distributed system is composed of multiple computers interconnected by a network. A distributed computation can be created by defining a collection of sequential computations executing in AUCs that coordinate their efforts by exchanging information. Coordination may be achieved by both the content of the information and the *time* at which it is exchanged (*synchronization* is one aspect of coordination). A distributed computation can execute as a collection of sequential computations in AUCs on a single, multiprogrammed computer, but in general the sequential computations that make up a distributed computation execute on AUCs in different computers in the distributed system. Sequential computations and AUCs remain the fundamental building block, and distributed computations extend them by adding means for the sequential communications to coordinate their individual actions using intercommunication mechanisms among the relevant AUCs.

It would be unusual for a distributed computation to be acceptable if it did not comply with the intent of the sequential computation rule. While it is sometimes acceptable for the two or more parts of the program to be executed in parallel (the Figure 2-5(b) scenario), the results from the distributed computation should be the same as if the distributed computation satisfied the sequential computation rule.² This is an important fundamental of distributed programming.

The sequential computation rule implies that the computation is **determinate**, meaning that each execution of the program (on the same input data) produces the same output results. If different executions of the computation produce different output results using the same input data, the computation is said to be **nondeterminate**. If the programmer violates the sequential computation rule, the program can still be made to be determinate, if the programmer used some other mechanism to assure determinacy.

Distributed programming is the art of writing programs that will be executed on a distributed system, generally as determinate programs. For a distributed program, the “program” is actually designed as two or more independent programs that each executes on an AUC as a sequential computation. The individual sequential computations are designed so that when they execute in concert, the resulting computation is determinate. The I/O instruction execution is perhaps the simplest example of this idea – recall the discussion in Section 2.2. In this case the two “distributed machines” are not AUCs, but are actually the processor and a device within a single computer, but the two components execute instructions independently and simultaneously. Figure 2-5(a) illustrated how the sequential computation rule can be followed by constructing the program so that it execute a wait-loop during the time that the device was executing the I/O instruction. That is, at any given moment, only one component of the computer is working on the *effective steps in the algorithm*, since the instructions in the processor wait-loop instructions do not contribute to any actual algorithmic processing – they only prevent the processor from working on the algorithm until the I/O instruction is being executed by the device.

In Figure 2-5(b) the situation is different since the computation explicitly violates the sequential computation rule: while the I/O instruction is being executed, the processor is executing other instructions in the algorithm (those in the `executeCodeFragment ()` function). If the instructions in the `executeCodeFragment ()` function happened to read or write the variable `x`, then the computation

² In a few cases, people have traded off the “correctness” of a distributed computation (in terms of its determinacy) if the resulting computation is much faster, and context of the computation assures ultimate correctness. But this is not typical – “kids don’t try this at home” until you understand exact conditions when it is acceptable for the computation to be nondeterminate.

would not be determinate. This follows because in some executions, the old value of `x` would be used in the `executeCodeFragment()` function, and in other executions the new value read by the input instruction would be used in the function; this is a known condition that causes nondeterminacy.³ How did the system know that the instructions that the processor would be executing would not be affected by the result of the I/O instruction execution? In this case the system cannot make that assurance; instead, the programmer took responsibility for that decision by carefully designing the code so that the code that executes on the processor simultaneously with the execution of the I/O instruction will be determinate.

Distributed programming generalizes this core idea by permitting overlapping execution of arbitrary instructions (in AUCs) on distinct physical computing units – whether they are within computers, or across computers in a distributed system. The general idea is to *partition the work of an algorithm into subalgorithms*, each of which can be executed relatively independent of its sibling subalgorithms. That is, a subalgorithm can be implemented by a small sequential computation (that follows the sequential computation rule), but at the same time, the unit of work that it performs contributes to the execution of the general algorithm. Subsequently each subalgorithm sequential computation can be assigned its own AUC when it is executed by the OS. The difficulty in creating this partition is in (1) defining subalgorithms that can execute relatively independently from one another, yet contribute to the solution of the general algorithm, and (2) ensuring that in the few cases where the subalgorithms interact with one another to coordinate their execution (for example to exchange information), that they do so in such a way that the logical pattern of sequential execution of the general algorithm is preserved.

3.2 Using the Network for Distributed Computations

Distributed computations depend on (AUCs to use) networks to exchange information among sequential computations executing at different machines on the network. In this section we will discuss how to write programs that use ISO OSI compliant networks (the vast majority of contemporary networks). Recall from [Section 2.1](#) that the physical and data link layers are typically implemented in the network device controller, called the NIC. The network layer and parts of the transport layer are usually implemented as part of the OS, so application programmers use the transport layer protocols (or higher) when they use the network. The dominant transport layer protocol is TCP. The TCP package enables applications to exchange information on a *full duplex byte stream channel* called a **connection** or **virtual circuit**. Byte stream I/O enables a sequential computation to write an arbitrary collection of bytes to the stream, and for another sequential computation to simply read the collection of bytes. For example, the POSIX.1 system call interface includes the functions `read()` and `write()` to receive and transmit bytes on a byte stream virtual device (the “device” could be a physical device, a file, a pipe, or another abstraction of an I/O device).

In a POSIX compliant system, you can write code as shown in the code fragment in Figure 3-4. In this example the code will write data onto a byte stream represented by the abstract/virtual device “MyOutputVirtualDevice”, and to read a bytes stream from the “MyInputVirtualDevice”. These are exactly the same system calls you would use to read or write bytes from/to a file; in POSIX-style I/O, the same system calls are used to read and write any virtual device that is implemented as a byte stream. In the example, the `open()` system call prepares the byte stream for I/O, and then the `read()` system call copies the number of bytes specified by the third argument, from the byte stream into the block of memory specified by the address in the second argument. And of course the `write()` system call copies bytes from the memory onto the byte stream. These are streamlined I/O functions in the sense that they do not handle nice features such as formatting data according to types, as does the `iostream` class in C++ or the `stdio` library in C. But of course the implementations of both of these libraries on POSIX class operating systems ultimately use the `read()` and `write()` system calls to read/write data from a byte stream.

³ For example, see Section 3.4 in [Nutt, 1992].

```
...
int inStream, outStream;
char buf[16]};
...
inStream = open("MyInputVirtualDevice", ...);
outStream = open("MyOutputVirtualDevice", ...);
...
/* Read 12 bytes from the input byte stream */
numRead = read(inStream, buf, 12);
...
/* Write 7 bytes to the output byte stream */
numWritten = write(outStream, buf, 7);
...
```

Figure 3-4: Code Fragment to Read/Write a Byte Stream Virtual Device

UDP is another network layer protocol, widely used for a different class of communication than TCP. UDP transmits/receives **datagrams** (variable-sized blocks of information). The TCP communication model was inspired by the notion of a telephone call: a caller establishes a connection (or virtual circuit) with a callee. Once the connection has been established, variable-length snippets of conversation can be transmitted across the circuit just as bytes can be transmitted across a byte stream. On the other hand, UDP follows a telegram model in which information is transmitted in a standalone block of information. There is no connection involved, just the idea that a sender can package information into a datagram and then send the datagram to a remote recipient.

TCP provides more service than does UDP: but with UDP the application program need not worry about establishing a connection at all, since datagrams do not use a connection. On the other hand, UDP does not enable byte stream operations – the transmitting program is responsible for collecting bytes into a datagram, and the receiving program must unpack the bytes from a program to interpret their values. Perhaps most importantly, by creating a mechanism to support a virtual circuit model, TCP is also able to assure that information that is transmitted using TCP will be delivered to the recipient. We say that TCP is a *reliable transport* protocol. On the other hand, UDP makes no such assurance: the UDP makes its best effort to deliver a datagram (if it is able to deliver a datagram, UDP does guarantee that its contents are the same as those that were transmitted), but it does not guarantee that the datagram will ever reach its recipient.

That is, TCP and UDP offer dramatically different transport layer protocols: TCP is reliable and byte stream oriented, while UDP is not guaranteed to be reliable (it depends mostly on the reliability of the underlying protocols) and it only delivers blocks of data in the form of datagrams. On the other hand, UDP transmission usually transmits less information over the subcommunication network than does TCP, so it has higher performance. Distributed computations that depend on never losing any data usually use TCP, but those that can compensate for missing data (and which require high performance) often use UDP. For example, a distributed system that supports financial transactions would probably use TCP, but a distributed system that distributes MPEG video streams among its constituent sequential computations would probably use UDP. (Many MPEG applications can be designed to operate, perhaps with degraded video quality, in cases where some data is lost during transmission.) Thus, an SCC multimedia player machine might use UDP to play a live video stream from a large file distribution server.

3.2.1 Programming the Transport Layer

Transport layer protocols provide *end-to-end* mechanisms for transmitting information from one sequential computation to another. A sequential computation that uses a transport layer protocol structures its communication with “peer” sequential computation that is also using the same transport layer protocol.

Generally, most of the details of the lower layer protocols used by the transport layer implementation are not visible to the programmer.

Programmers can choose to use connection-based protocols (like TCP) or packet-based protocols (such as UDP). In either case, transport layer services will use its own address space for referencing AUCs around the network. The transport layer address space is extended beyond internet addresses so that transmitting party can reference a specific *port* on a *remote host* on a *remote network*. That is, the network layer identifies senders and receivers in terms of a $\langle net, host \rangle$ pair. IP will route a packet from a specific *host* computer on a particular *net*, to a *remote host* computer on a *remote net*. But IP does not distinguish among any of the different sequential computations that might be executing on either the transmitting or receiving computer. The transport layer defines a collection of ports for each host computer. Each sequential computation on a given *host* on a given *net* can have a particular *port* that it uses for its network communication. For example web servers always use port 80 to transmit and receive information. The actual implementation of the port depends on the OS: in the Berkeley System Distribution (BSD) Unix system, the **socket** OS data structure is associated with a port ... but more on that later.

We have already described the differences and similarities of datagrams and byte streams. These two transport layer data types are the work horses, but they can also be extended by higher layer protocols to support various types of network communications, including formatted messages, remote file service transport, remote procedure calls, remote method invocation for object-oriented environments, and so on.

Transport Layer Addresses

Network layer internet addresses are of the form $\langle net, host \rangle$. Each ordered pair uniquely identifies a particular host computer that is connected to a particular network in an internet. IP has been used for network communication for many decades (since the 1970s). It has passed through many versions. Today, IPv4 (IP Version 4) is still widely used in the Internet, although it was updated to (and is gradually being replaced by) IPv6 in the mid 1990s. IPv6 increases the size of the internet address space (from 32-bit to 128-bit addresses), as well as adds various features.⁴ IPv4 encodes the net and host addresses in a 32-bit word, using one of 4 possible formats (called Classes A, B, C, and D).⁵ For example, in a Class C address format, the 3 most significant bits are set to “110” to tag the address as a Class C address. The next 21 most significant bits are a binary network number, and the 8 least significant bits designate a host on the specified network. An IP address such as 192.168.0.12 (this is the form of the 32-bit address in the figure uses the “dotted decimal notation,” where each of the four decimal numbers represents a byte in the address) refers to a 32-bit byte 0xc0a8000c. That is, 192.168.0.12 can be decoded as follows: the most significant 8-bits are represented by 192₁₀ which corresponds to the hexadecimal representation of c0₁₆, which in turn is the 8-bit number 11000000₂. The 168 represents the next 8 bits, and is represented as in hexadecimal as a8₁₆ which has the bit pattern 10101000₂. Of course the zero decimal is 00000000₂. The least significant 8 bits is represented by the decimal number 12₁₀, the hexadecimal number, 0c₁₆, and the binary number 00001100₂. That is, the dotted decimal notation number 192.168.0.12 represents the 32 bit pattern

```
11000000 10101000 00000000 00001100
```

Regrouping the bits to match the Class C IP address format, we have

```
110 0000010101000000000000 00001100
```

Where the 3 most significant bits identify the address as a Class C address, the next 21 bits specify a net of address of 00a800₁₆, or 43008₁₀. The least significant byte specifies a host address of 12₁₀. Which represents the 32-bit Class C IP address of $\langle net, host \rangle = \langle 43008, 12 \rangle$.

A port is a third component of a transport layer address, meaning that a full transport layer address has the form $\langle \langle net, host \rangle, port \rangle$. Each machine can support a number of ports to represent places for information to be sent and received from/to software in the machine. Figure 3-5 represents a host machine

⁴ Please browse the Web for current definitions and descriptions of IPv6. You may find it useful to start at <http://www.ipv6.org/>.

⁵ See [Stevens, 1994], particularly Section 1.4 for more details.

at its unique network layer $\langle \text{net}, \text{host} \rangle$ address, along with n internal ports. This figure illustrates how the 3-component network address ($\langle \langle \text{net}, \text{host} \rangle, \text{port} \rangle$) can be mapped to endpoints that can then be used by an AUC on the given host machine. That is, the transport layer OS software creates a set of communication **ports**, p , within the machine. If a sequential computation wants to use the network layer to communicate with a peer sequential computation, its AUC uses a port to transmit the information, and it directs outgoing information to a remote $\langle \langle \text{net}, \text{host} \rangle, \text{port} \rangle$ that can be used by the receiving sequential computation on the remote machine. Implicit in this description is that ports are OS resources, and an AUC must obtain the right to use a particular port in behalf of a sequential computation, either by dynamically requesting the port, or by having the port permanently reserved (for example port 80 is permanently reserved for the use of use of the higher layer HTTP protocol when it uses the network layer for web-based communication).

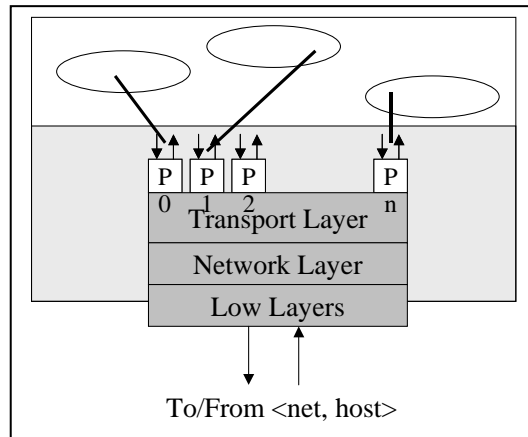


Figure 3-5: Extending the Address Space with Ports

BSD Sockets

A BSD **socket** is an OS data structure that can be dynamically associated with a port (see Figure 3-6). The socket is designed so that a sequential computation can configure it to connect to the port, and then to transmit and receive information using a variety of transport layer protocols, including TCP and UDP. Thus the OS socket facility exports a set of system calls that enable sequential computations to manipulate sockets. In operating systems that use the BSD socket package (including Linux and Windows), programmers use sockets, rather than ports, as endpoints in a network communication.

With BSD sockets, a sequential computation prepares to use the network by first requesting a socket from the OS using the following system call:

```
int socket(int domain, int type, int protocol);
```

The `domain` argument specifies whether the socket should direct the supporting network layer to be configured to use IPv4 or IPv6 with communication done over this socket. The `type` argument is used to configure the socket to use streams/connections or datagrams (and a few other choices). The third argument selects a `protocol` to use with the socket; it is safe to set the `type` field and to use a value of zero in the `protocol` field. The OS will then select the default protocol for the given socket type, for example Linux systems use TCP with stream sockets, and UDP with datagram packets. A successful `socket()` call returns an integer *handle* to reference the allocated socket. You can think of a handle as a small integer that the OS uses as an index into an internal table of socket descriptions.

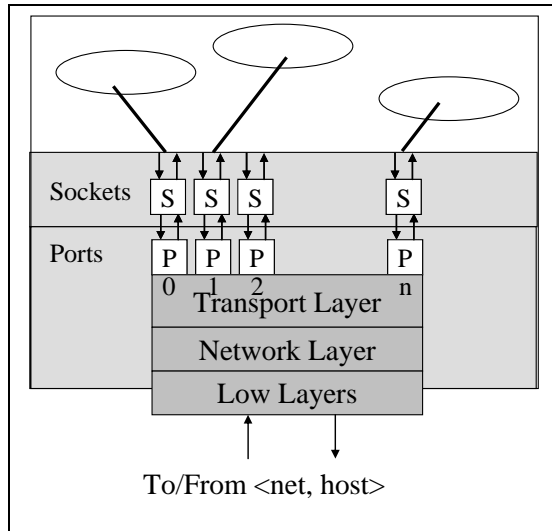


Figure 3-6: The BSD OS Socket Package

The `socket()` call is analogous to an `open()` call for a file or device in that it sets up OS data structures that enable the sequential computation to use the handle in standard OS `read()` and `write()` system calls to send and receive information over a network. By creating a socket, the program has been assigned an OS resource that can implicitly or explicitly associated with a port, thereby enabling the sequential computation to send and receive information (via the socket) over the network. You can use the `bind()` system call (described below) to explicitly associate the socket with a particular port, or if you do not intend for anyone else to send information to you via a particular `<<net, host>, port>`, then there is no need to explicitly associate the socket with a port (and no need to use the `bind()` system call), since the OS will simply choose one for you.

3.2.2 Using UDP with BSD Sockets

To send information using the transport layer, you will need to know the recipient's address – the `<<net, host>, port>` it will use to accept information on its socket. First, you must know an external host name (such as the host names found in the `/etc/hosts` file on a Linux machine, or a name registered with a Domain Name Service registrar). By default, every such file contains an entry for the `localhost` name, usually with the value `127.0.0.1` (a class A IP address). Since the `/etc/hosts` file normally has global read permission, you can view its contents on your local machine to see external host names that you can use with `gethostbyname()` to create a `<net, host>` pair. Let's suppose that you know such a name, and that you have stored it in a string named `destHostName`. Here is a Linux code fragment that creates a data structure containing a transport layer address:

```
#include      <netinet/in.h>
#include      <sys/socket.h>
#include      <netdb.h>
...
char destHostName[HOSTNAMELEN];
struct hostent *host;
struct sockaddr_in dest;
...
host = gethostbyname(destHostName);
bzero(&dest, sizeof(dest));
dest.sin_family = AF_INET;
dest.sin_port = htons(port);
```

```
bcopy(host->h_addr, & dest.sin_addr, host->h_length);
...
```

The three `#include` files contain data structure definitions and function interfaces that are used in the code fragment. The fourth line of code declares the `destHostName` string, and the following two lines declare the host data structures to hold the internal form of the `<net, host>` pair, that will be used to construct the complete transport layer address in the `dest` data structure. The call to `gethostbyname()` translates the external address name, `destHostName`, into an internal `<net, host>` pair. The code fragment uses `host` to point the resulting `struct hostent` data structure that contains the `<net, host>` pair. The `bzero()` function sets all of the fields of the `struct sockaddr_in dest` data structure to zero. The next line sets the `dest.sin_family` field to the type of the address being defined in the data structure – there are a few choices, although the dominant type is `AF_INET`, for an IP address. The next line assigns the port number to the `port` field in the `dest` data structure. This code presumes an integer value of port has been defined prior to its use. The `htons()` function call translates the form of the port number from the host machine representation to a standard network format. Finally, the `bcopy()` function copies the IP address from the `host->h_addr` field to the `dest.sin_addr` field in the `struct sockaddr_in dest` data structure. Now, the `dest` data structure can be used as an IP-compliant (`AF_INET`) transport layer address.

As an illustration of the use of a distributed computation and the use of BSD sockets, we can redesign the simple byte stream file copy example shown in Figure 3-2, and represented by pictorial representation in Figure 3-7(a) into a distributed computation (Figure 3-7(b)) with reader and writer sequential computations. That is, we have split the original sequential computation into two sequential computations, running on two different AUCs on two different machines. Then we use the BSD socket model with the transport layer to transfer the file contents across the network. In effect, we will have created a means of copying a file from one machine to a remote machine.

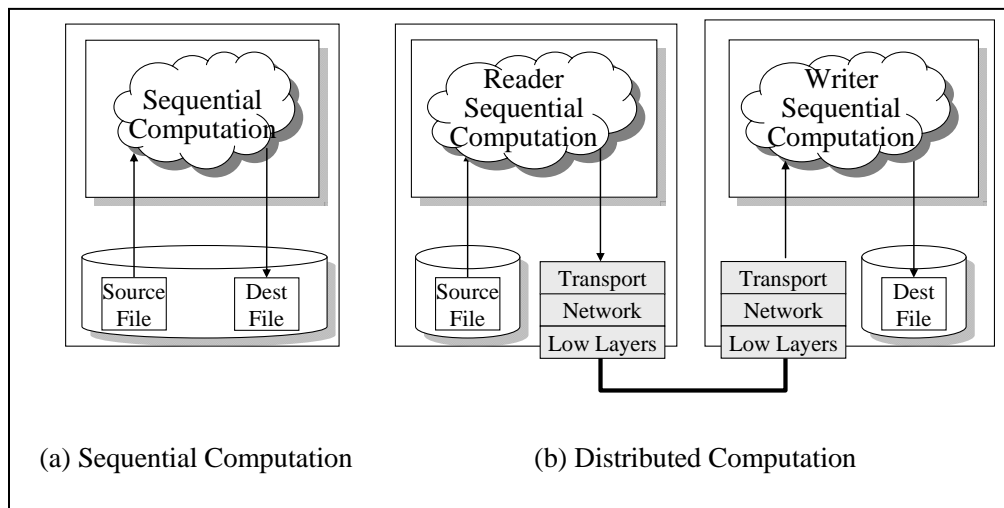


Figure 3-7: A Distributed Computation to Copy a File

Figure 3-8 shows the reader sequential computation code skeleton and Figure 3-9 shows the writer sequential computation code skeleton that, together, implement the distributed computation shown in Figure 3-7(b) – you can also see similar, working code elsewhere.⁶ The new code in these skeletons is the `sendto()` system call in the reader, and the `bind()` and `recvfrom()` system call in the writer. Let's start discussing the code by first discussing the `bind()` call.

⁶ For example, see Chapter 6 of [Stevens, 1990], or supplementary materials for this book.

The `socket()` call does not associate the socket with any particular port number; if the socket is subsequently *used* without specifying a port association, the OS will perform its own implicit association when information is sent over the net, but it will not necessarily let the sequential computation know what port number it is using. The sequential computation can specify which port number it wants to associate with the socket using the `bind()` system call:

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

The `sockfd` argument is the handle created by the `socket()` call. The `my_addr` data structure specifies the `<<net, host>> port>` that is to be associated with the socket, where the `addrlen` field specifies the length of the `my_addr` field. This last argument is necessary since `bind()` works with a number of different types of socket addresses. (The `struct sockaddr_in` type is one choice of union defined by the `struct sockaddr` type; in OO languages, we would probably say that `sockaddr` was a base class for `sockaddr_in`.)

```
/* Set up a UDP socket to talk to the server */
    skt = socket(AF_INET, SOCK_DGRAM, 0);

/* Build the server's address */
    host = gethostbyname(writerHostName);
    bzero(&writer, sizeof(writer));
    writer.sin_family = AF_INET;
    writer.sin_port = htons(port);
    bcopy(host->h_addr, &writer.sin_addr, host->h_length);

/* Run the reader part of the distributed computation */
    (See Figure 3-2)

/* Here is the main reader loop */
    while((numRead = read(inFid, buf, BUF_LEN)) > 0) {
/* Now send a datagram to the writer code */
        sendto(skt, buf, numRead, 0,
              (struct sockaddr *) &writer,
              sizeof(struct sockaddr));
    }
/* Send the EOF signal */
    buf[0] = EOF_CHAR; // This will be a 1-byte datagram
    sendto(skt, buf, 1, 0,
          (struct sockaddr *) &writer,
          sizeof(struct sockaddr));

/* Clean up and quit */
    printf("reader: Terminating ...\n");
    ...
    exit(0);
}
```

Figure 3-8: Code Skeleton for the UDP Reader Sequential Computation

In this version of the code the reader uses the `sendto()` system call to transmit a datagram to the writer (using UDP). In Figure 3-8 the main loop first reads `BUF_LEN` characters from the input file; the buffer into which `read()` places the data is then passed to `sendto()`, which will copy the `numRead` bytes of data from `buf` into a datagram, and then send it to the `<<net, host>> port>` that was built into the `struct sockaddr_in` `writer` data structure. When the reader encounters an end-of-file condition on the input file, it exits the loop, writes an `EOF_CHAR` into a datagram, and then send it to the writer. If

the reader does not do this, the writer will not know that the reader is not going to transmit any more datagrams to it; we have to create our own “end-of-file” signal between the reader and writer.

In the writer sequential computation (Figure 3-9), the writer creates its own version of its transport layer address so that it can call `bind()` to associate the writer’s socket with the designated `<<net, host>`, `port>`. Thereafter, the reader can send datagrams to the specified `<<net, host>`, `port>`, and they will be delivered to the writer’s socket. The writer receives datagrams using the `recvfrom()` system call. This system call copies the contents of a datagram into the buffer, `buf`, so that it can be copied to the writer’s output file using a conventional `write()` system call.

```

/* Set up a UDP socket to talk to the writer */
    skt = socket(AF_INET, SOCK_DGRAM, 0);

/* Build the writer's address */
    host = gethostbyname(writerHostName);
    bzero(&writer, sizeof(writer));
    writer.sin_family = AF_INET;
    writer.sin_port = htons(port);
    bcopy(host->h_addr, &writer.sin_addr, host->h_length);

/* The socket has to be bound to an IP address to receive data sent
 * to that IP address
 */
    if(bind(skt, (struct sockaddr *) &writer, sizeof(writer))) {
        fprintf(stderr, "Bind error, restart ...\n");
        exit(1);
    }

/* Run the writer part of the distributed computation */
(See Figure 3-2)

    while(1) {        // A value of "1" is TRUE
        numRcvd = recvfrom(skt, buf, BUF_LEN, 0,
            (struct sockaddr *) 0, (int *) 0);
        if(buf[0] == EOF_CHAR)
            break;
        write(outFid, buf, numRcvd);    // Write to the file
    }

/* Clean up and quit */
    printf("writer: terminating ...\n");
    ...
    exit(0);
}

```

Figure 3-9: Code Skeleton for the Writer Sequential Computation

3.2.3 Using TCP with BSD Sockets

As mentioned earlier, the telephone analogy is used by the TCP to implement the connections (or virtual circuits). If two sequential computations agree to establish a virtual circuit between them, then either one of them can transmit a byte stream across the virtual circuit without being concerned about packet boundaries. And as pointed out earlier, TCP *guarantees* that all packets used to hold the byte stream will be delivered in the order they were sent.

Opening a virtual circuit requires that the sender and the receiver agree to exchange information. As described previously, any sequential computation intending to communicate with other sequential computations must establish a port so that other processes have an endpoint – like a telephone number – on which to connect the virtual circuit. After both processes have created a port, one of the AUCs (we have arbitrarily chosen the reader) must establish the virtual circuit by setting up its own endpoint. It then

requests that the remote end accept the request to connect the virtual circuit to the specified communication port on behalf of the receiver. When a pair of sequential computations have completed their use of the virtual circuit, they must “tear it down,” since network resources are required to keep the virtual circuit intact.

TCP is the prevailing transport layer implementation in contemporary networks. It provides virtual circuit capabilities that enable a sending process to establish a virtual circuit to a remote machine and to exchange information bidirectionally over the connection. Communication using TCP is reliable, so TCP has become the workhorse protocol for contemporary network applications. It is used in window systems (including the X windows system), remote file systems, and mail systems.

```

/* Set up a UDP socket to talk to the writer */
    skt = socket(AF_INET, SOCK_STREAM, 0);

/* Build the writer's address */
    (See Figure 3-8)

/* Open the connection to writer */
    if(connect(skt, (struct sockaddr *) &writer,
              sizeof(writer)) < 0) {
        fprintf(stderr,
            "reader: connect() failed, halting\n");
        exit(1);
    }

/* Run the reader part of the distributed computation */
/* Open the source file */
    (See Figure 3-2)

/* Here is the main reader loop */
    while((numRead = read(inFid, buf, BUF_LEN)) > 0) {
        /* Now send a datagram to the writer code */
        write(skt, buf, numRead);
    }

/* Clean up and quit */
    printf("reader: Terminating ...\n");
    ...
    exit(0);
}

```

Figure 3-10: Code Skeleton for the TCP Reader Sequential Computation

Figure 3-10 contains the TCP reader code, and Figure 3-11 contains the companion writer code. As before, both blocks of code create a socket so that they can use the transport layer network. Next, they both create an internal form of the <<net, host>, port>, the `struct sockaddr_in writer`, which the reader will use to request a connection to the writer. This connection will later be used to transmit the file contents from the reader AUC to the write AUC.

```

/* Set up a UDP socket to talk to the writer */
   skt = socket(AF_INET, SOCK_STREAM, 0);

/* Build the writer's address */
   (see Figure 3-9)

/* The socket has to be bound to an IP address to get data sent
 * to that IP address
 */
   if(bind(skt, (struct sockaddr *) &writer, sizeof(writer))) {
       fprintf(stderr, "Bind error, restart ...\n");
       exit(1);
   }

/* Set up the socket to listen for connect requests */
   listen(skt, MAX_REQUESTS);
   newSkt = accept(skt, (struct sockaddr *) &writer, &addrLen);
   close(skt);      // Close the listener socket, skt
                   // Use the newSkt

/* Run the writer part of the distributed computation */
/* Open the destination file */
   (See Figure 3-2)

   while((numRcvd = read(newSkt, buf, BUF_LEN)) > 0) {
       usleep(1000);
       write(outFid, buf, numRcvd);
   }

/* Clean up and quit */
   printf("writer: terminating ...\n");
   ...
   exit(0);
}

```

Figure 3-11: Code Skeleton for the TCP Writer Sequential Computation

Let's suppose that the writer AUC executes before the reader AUC. After it has bound its socket to the agreed upon transport layer address, it calls `listen()` to prepare to accept connection requests, and then it calls `accept()` to wait for a remote AUC to request a connection. The write is suspended at the `accept()` call until some other AUC issues a connection request.

When the reader AUC begins to execute, it calls `connect()`, which will issue a connection request to the designated `<<net, host>, port>` – in this case, the writer AUC. Since the writer has already begun execution, it is blocked at its `accept()` call. Hence, the reader AUC's `connect()` call contains code that interacts with the writer AUC's `accept()` call, causing the transport layer software for the two machines to jointly create a connection between the two AUCs. After the connection has been established, the reader continues executing code following its `connect()` call, while the writer continues executing code (simultaneously) following its `accept()` call.

Next, the reader and writer each open their respective files – the same code as in both previous examples. Then both the reader and writer enter loops to copy the file contents from the machine running the reader AUC to the machine running the writer AUC. The `read()` and `write()` POSIX system calls can be used to read and write the TCP byte stream. So the reader just reads its source file (until it encounters the end of the file), transmitting each “gulp” of bytes to the writer by using the `write()` system call on the socket. That is, for TCP, writing to a socket looks no different than writing to a file. Meanwhile, the writer AUC reads the “gulps” of bytes from its socket, then writes them to its local file.

When the reader finishes, it closes the socket, thereby issuing an end-of-file indicator to the writer AUC. Thus the writer simply reads the socket until `read()` sees an end-of-file indicator, then it closes its end of the connection, closes the file it just created, and quits.

3.3 The Client-Server Model Revisited

In the previous section, the reader AUC and the writer AUC needed to create a connection between them in order to transmit bytes from the reader to the writer. This is analogous to two people wanting to converse using the telephone, but one of them must take a passive role in which they wait for the other to proactively place the telephone call. Informally, we refer to the person who waits as the “server,” and the one who places the call as the client. If you reconsider the code in Figure 3-10 and Figure 3-11, it is obvious that the reader performs different actions than does the writer in order to establish a connection. That is, the writer AUC started first, and ultimately blocked on an `accept()` system call to passively wait for the reader to actively *establish* the connection with a `connect()` system call. For the explicit task of establishing a connection, the writer AUC plays the passive **server** role introduced in [Chapter 2](#), while the reader AUC plays the active client role. Notice that the reader and writer only play the client and server roles to perform a task – until the connection has been opened. Then either sequential computation could play either role.

This active/passive role situation occurs so frequently in distributed programming that it has come to be recognized as a standard paradigm for communication, and is called the **client-server model** for describing how two AUCs should interact. In the discussion in Chapter 2, client-server computing was motivated by talking about how a device implements I/O instructions, and then by seeing how parts of a sequential computation can be extracted from the original computation and implemented in a remote component (first as a device, then as an SCC with a device). The device, or the SCC plus device, is a server that waits for the client application program to request an I/O service. As suggested by the connection protocol, the idea can be generalized to refer to other blocks of computation.

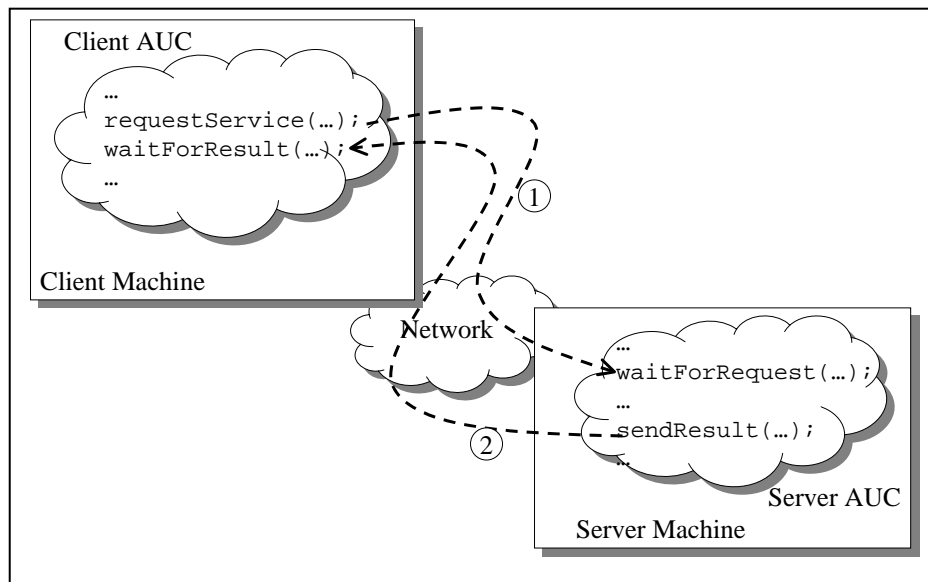


Figure 3-12: Client-Server Communication

The general pattern for client-server communication is summarized in Figure 3-12. The server AUC is assumed to be running continuously on the server. When there is no pending service request, the server AUC waits for a client to request service (see `waitForRequest()` in the figure). Meanwhile when a client AUC decides that it desires service, it sends a network message to the server requesting that it perform a designated service (see `requestService()`— see Step 1 in the figure); the client ultimately waits for a result from the server (at the `waitForResult()` call). The server AUC eventually gets a service request (at `waitForRequest()`) causing it to perform the specified service. Once the service has been completed, the result is returned to the waiting client (using the `sendResult()` call).

3.3.1 Example: The Smart Thermometer

Suppose that an SCC is to implement the “smart thermometer,” and the ST OS provides transport layer communication facilities such as the BSD socket package, for example TCP. Then the SCC can read the control path from a remote host machine to the SCC (using the transport layer), read the thermometer device (using I/O commands), and it can send information to the large host machine using the transport layer services. (See Figure 3-13, which is just a copy of Figure 2-9). Then the SCC could conceivably do all of the work required of it by executing a single sequential computation that has the code skeletal shown in Figure 3-14.

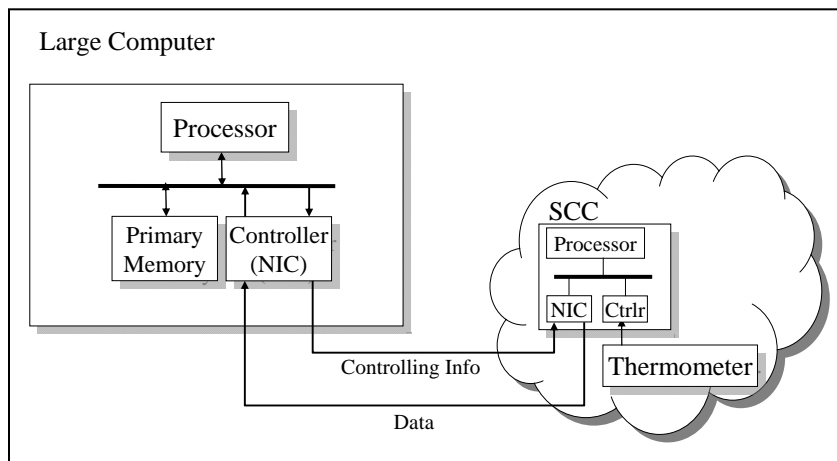


Figure 3-13: A Device Controlled by a SCC

This algorithm works fine, provided that the large machine does not need to provide new control parameters once the program has been initialized (for example to reset the device). If it were deemed necessary, the SCC OS might need to provide additional services to the application, for example a function to *poll* a device to see if it has pending data to be read. This new option is shown in Figure 3-15: the sequential computation polls the network once each time through the loop to see if there is any new control data from the large computer. If there is data, then it is read from the socket, then used to change the parameters that control the SCC’s behavior.

Obviously, we could continue to add more little tasks, like checking the network for input data, by just writing another code fragment into the basic SCC sequential computation loop. As we add more and more little tasks, it takes longer and longer to complete the loop, so it will be longer and longer between times that any particular little task is performed. At some point, we would probably decide that it would be necessary to do things differently in order to provide timely service to little tasks that need attention. This is hard to do in an ST class computer, but easy to do in a machine that has interrupts. This suggests that as we burden an ST class computer with too much work, we need to think about upgrading the software model, hardware, and operating system to a multithreaded (MT class) SCC with interrupts ... but that is another story, for another booklet.

```

/* The Control Path is data flowing from the large computer to the
 * SCC via the TCP connection.  Thermometer data flows
 * from the SCC to the large computer over the same TCP
 * connection.
 */
skt = socket(...)'
...
read(skt, high_threshold_value); // Read control info from host
read(skt, low_threshold_value); // Read control info from host
while(...) {
    ...
    read(THERMOMETER, value);
    if(value > high_threshold_value){
        write(skt, value); // Write data to the host
        continue;
    }
    if(value < low_threshold_value){
        write(skt, value); // Write data to the host
        continue;
    }
    ...
}
...

```

Figure 3-14: Basic Code Skeleton for Smart Thermometer

```

/* The Control Path is data flowing from the large computer to the
 * SCC via the TCP connection.  Thermometer data flows from the SCC
 * to the large computer over the same TCP connection.
 */
skt = socket(...)'
...
read(skt, high_threshold_value); // Read control info from
host
read(skt, low_threshold_value); // Read control info from host
while(...) {
    ...
    read(THERMOMETER, value);
    if(value > high_threshold_value){
        write(skt, value);
        break;
    }
    if(value < low_threshold_value){
        write(skt, value);
        break;
    }
    if(poll(skt)) {
/* There is data waiting at the network socket)
        read(skt, new_ctl_data);
        // Handle new control data
        ...
    }
    ...
}
...

```

Figure 3-15: Refined Smart Thermometer Example

This example has an interesting twist to it in terms of distributed computation: it is clear from everything that we have learned in Chapters 2 and 3 that there are two sequential computations working simultaneously on two different computers (the large computer and the SCC in Figure 3-13). By using the simplest form of communication under the TCP model, the system behaves as a true distributed computation with overlapping execution of the two sequential computations! If we wanted the two machines to execute as if they were a sequential computation, then we would have to add more functionality to the system. If we wanted to ensure that the distributed computation was determinate, we can do less work, but we may have to think harder to be sure that the system really does what we think (hope?) it is going to do. What do you think: is the distributed computation based on the SCC sequential computation shown in Figure 3-15, and presuming matching reads and writes on the large computer determinate or not?⁷

3.4 Remote Procedures

For several decades, procedures have been used to modularize computation in a sequential program. Today, professional programmers design software so that modules (or classes) encapsulate data and function implementations behind public interfaces. A basic semantic rule for these subcomputations is that they execute sequentially with respect to the computation that invokes them – they preserve the sequential computation rule when they are invoked by a sequential computation.

As we have seen when we learned about network communication, it is generally easier to break the sequential computation rule across the distributed system than it is to conserve it. Modularization can also be used to hide implementation in distributed programming, since the client-server interface is analogous to a procedure or class interface. In distributed programs modularization can also enable location transparency and scheduling, meaning that programmers don't have to know anything about where a server AUC is located or how it is scheduled to run. The cost is that the programmer *does* need to learn a new environment, such as the BSD socket/TCP/UDP environment, in which to construct application programs if they wish to take advantage of the underlying distributed hardware. The **remote procedure call (RPC)** paradigm is the dominant extension to sequential programming environments that takes advantage of networks. Interestingly, RPC explicitly preserves the sequential computation rule across distributed computations. It does this by making every interaction between distributed AUCs behave as if it were a local procedure call – if a client asks a server to do some work, then the client suspends its own operation until the server responds with a result.

3.4.1 How Does RPC Work?

RPC is a network protocol that allows software on one machine to literally invoke a procedure that is implemented on another machine. On the calling machine, the RPC facility will intercept a conventional procedure call, and then pass the call and parameters to the remote machine. Thinking about this abstractly, RPC is really just a specialized pattern of client-server communication in which a client AUC performs a send operation immediately followed by a read operation that waits until the server AUC completes the service requested of it. As in all client-server computations, the server blocks until it receives the message containing the RPC request that is sent by the calling AUC's system software. The server then performs the service and returns a result by sending it to the client AUC.

This control flow/synchronization paradigm is summarized in Figure 3-16. For comparison, Steps 1 and 2 illustrate a conventional local procedure call: In Step 1 the calling procedure pushes the arguments onto the stack, saves the return address, then jumps to the entry point for `localF()`. When `localF()` completes its work, it returns to the calling procedure, and the calling procedure continues.

⁷ To the extent that the large computer complies with the sequential computation rule with respect to the smart device, it is determinate. This is because the SCC strictly adheres to the sequential computation rule in its program.

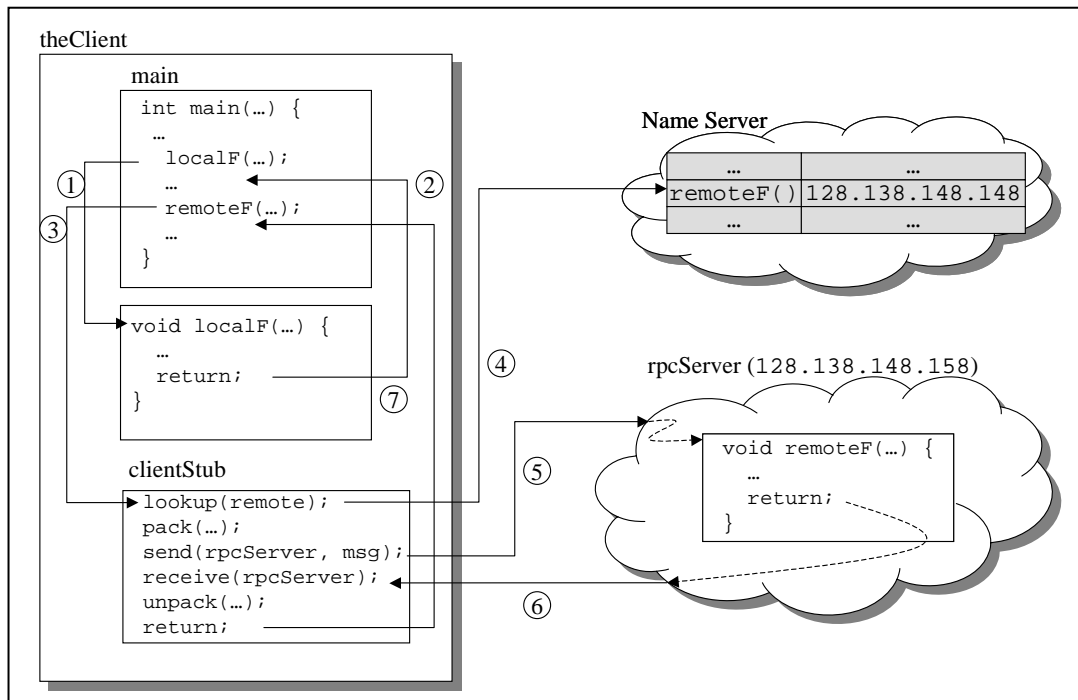


Figure 3-16: Remote Procedure Call Organization

Step 3 is a remote procedure call – the program calls a client stub procedure rather than the actual remote procedure. The **client stub** is a small procedure that first looks up the IP address of the machine that implements the remote procedure. This is done by sending a lookup request to a name server (Step 4) that contains a table of remote procedure names and the IP address of the machine that can execute the remote procedure. In a conventional procedure call, the main program pushes the arguments onto the stack and calls the procedure. In the client stub, the arguments are packed into a message (UDP packet or TCP structure), then transmitted to the RPC server (Step 5). Once the RPC request has been issued to the server, the client blocks, waiting for a result from the RPC server. Hence, the caller is idle while the called procedure executes in both the conventional (Steps 1 and 2) and RPC (Steps 3-6) paradigms.

Conceptually the RPC server, `rpcServer`, will have blocked waiting for an RPC request. When an RPC message arrives (Step 5), the server unpacks the arguments. In the conventional case, this is analogous to retrieving the parameters from the stack. The server then determines the name of the procedure and calls it. Once the procedure has completed executing, it returns to the main program in `rpcServer`. The main program packages up return parameters and notifies the `theClient` of the call completion (Step 6). The client stub then returns to the calling program in Step 7.

The RPC mechanism enables two processes to interact with one another using the control flow paradigm from conventional procedure calls. The RPC facility allows a programmer to write calling and called application procedures and then to execute the caller procedure in one process and the called procedure in a remote process on another machine without the programmer's knowing any details of messages or networks. From Figure 3-16, it is apparent that the RPC paradigm is a structured set of message sends and receives. Hence, RPC is normally thought of as a high-layer network (or possibly a session layer) protocol.

3.5 Summary

Computations are based on the idea of a sequential computation. Distributed computations are collections of interacting sequential computations. Since the sequential computations can be implemented in a physically distributed system, it is necessary the sequential computations to have a network at their disposal. Contemporary client-server computing makes heavy use of the ISO OSI transport layer to implement datagram (UDP) and connected (TCP) communication across sequential computations running in their own AUCs on distinct machines. The transport layer uses its own address space (of the form <<net, host>, port>) to identify distributed AUCs. The BSD UNIX socket packet is the dominant means for adapting the OS communication models to the transport layer protocols.

Remote procedures are another workhorse tool for distributed computing, especially SCC computing. This facility allows an SCC to invoke compute intensive and data intensive services that are implemented on a large network-accessible machine without special concern about determinacy and adherence to the sequential computation rule.

3.6 References

1. Nutt, Gary J., *Centralized and Distributed Operating Systems*, Prentice Hall, 1992.
2. Stevens, W. Richard, *UNIX Network Programming*, Prentice Hall Software Series, 1990.
3. Stevens, W. Richard, *Advanced Programming in the UNIX® Environment*, Addison Wesley Professional Computing Series, 1992.
4. Stevens, W. Richard, *TCP/IP Illustrated, Volume 1: The Protocols*, Addison Wesley Professional Computing Series, 1994.