THE DISTRIBUTED SOFTWARE CONCEPT OF REMOTED PROCESSES, THEIR APPLICATION AND IMPLEMENTATION

Wu-Hon Francis Leung

Wu-Hon Francis Leung *is a supervisor in the Exploratory Switching Networks Department at AT&T Bell Laboratories in Naperville, Illinois. His group does applied research on integrated multimedia (voice, data and image) services, packet network services, protocols and network management software, all based on wideband packet technology. Mr. Leung joined AT&T in 1978. He has a B.S. in electrical engineering from California State University, Northridge, and both an M.S. and Ph.D. in computer science from the University of California at Berkeley.*

The notion of remoted processes is a software concept for distributed systems, such as AT&T's 5ESS™ switching system. When a process is remoted, some of its programs are placed and run in a processor other than the one that first created it. Remoted processes are supported by a remoting subsystem. It permits the programmer to separate the program space of a remoted process and put parts of it into different processors without affecting the design of that process or of other processes that interface with it. This allows flexible placement of programs for the switch's call-processing features into different control processors. Without causing programming changes, these programs can be shifted to other processors according to expected feature usage and when the real-time and address-space capacity of the processor technology changes in specific generics. This paper describes the implementation of the essential features of the 5ESS switch remoting subsystem, including the techniques used for software auditing and exception handling.

Motivation and Basic Approach

The concept of a *process*, a sequential computation that can be executed asynchronously with other processes, is a design abstraction for dealing with the complexity of concurrency in real-time systems. Concurrency means several different tasks are running in parallel, controlled by the same operating system and often sharing system resources.

A process is considered a unit of concurrency, and the collection of processes in a real-time system may then model the concurrency that is inherent in the application. For instance, in call-processing applications,[1] a separate process can control each telephone terminal.

By exchanging messages or running some synchronization primitives, processes can communicate with each other so that they can cooperate to do some system function. Message passing and synchronization primitives are system subroutines of the operating system.

Structurally, a process is a collection of procedures that includes an *entry procedure,* the first one executed when the process is invoked. The *program space* of the process consists of all procedures that the entry procedure transitively calls.

Earlier work on software for distributed systems typically assumed that the program space for a process had to fit within a single processor. (A distributed system is a collection of self-contained processors, each with its own memory, interconnected by some communications medium.) Thus, if two procedures reside in different processors, they cannot call one another and must be executed by two different processes.[2,3] Consequently, when one procedure must initiate the other's operation and must know the result of the operation (these are the essential semantics of a procedure call), then some communications protocol must be designed into the two processes to allow that to happen.

This restriction often gets in the way of the logical design of processes, especially when the system uses microprocessors. Because of processor-technology constraints on real-time capacity and address space, procedures that logically belong to a process might have to be placed in different processors. As a result, the software must be artificially fragmented into different processes.

The traditional way to solve the address-space constraint is to use a memory-management subsystem, placing some programs in secondary storage. However, when the response-time requirement is very stringent, such as the milliseconds requirement in telecommunications call processing, this approach is too slow.

This paper describes a software subsystem, called a *remoting subsystem,* that allows us to place a subset of a process's procedures in a processor other than the one in which the process is first created. We call this subset of procedures *remoted procedures,* and say a process is *remoted* if it contains remoted procedures. The mechanics of moving those procedures to another processor is called *remoting.*

The essential requirements of remoting are that it does not require program changes either in the remoted process or in processes that communicate with it. As a result, when remoted procedures are moved back to the processor where the remoted process is first created, the process (which is now *unremoted*) will function properly without program change.

A remoting subsystem was first used successfully in AT&T's 5ESS switch, a distributed computer system for telephone switching applications.[1,4-6] Many major call-processing features, such as operator and coin services, were implemented as remoted processes in the first issue of the programs that control this switch. Because of the remoting subsystem, these call-processing features can be placed in any of the switch's processors without program change.

Thus, the remoting subsystem permits us to tailor software placement to individual office needs for memory and switching capacity. To a certain extent, it relieves the programmers of concern with hardware constraints, such as memory space, so they can concentrate on the design's logic.

More recently, Nelson[7] and Spector[8] have discussed communication subsystems that enable some processes in a distributed computer system to make *remoted references* to objects, including procedures, in other processors. But, the scope of their work differs significantly from ours. For instance, neither of them considers how to preserve the communications interface if the remotely executed procedure contains interprocess communications primitives.

In the sections that follow, we discuss the environment in which our remoting subsystem is implemented and how we preserve the interfaces, both internal and external, to remoted processes. We also describe tech-

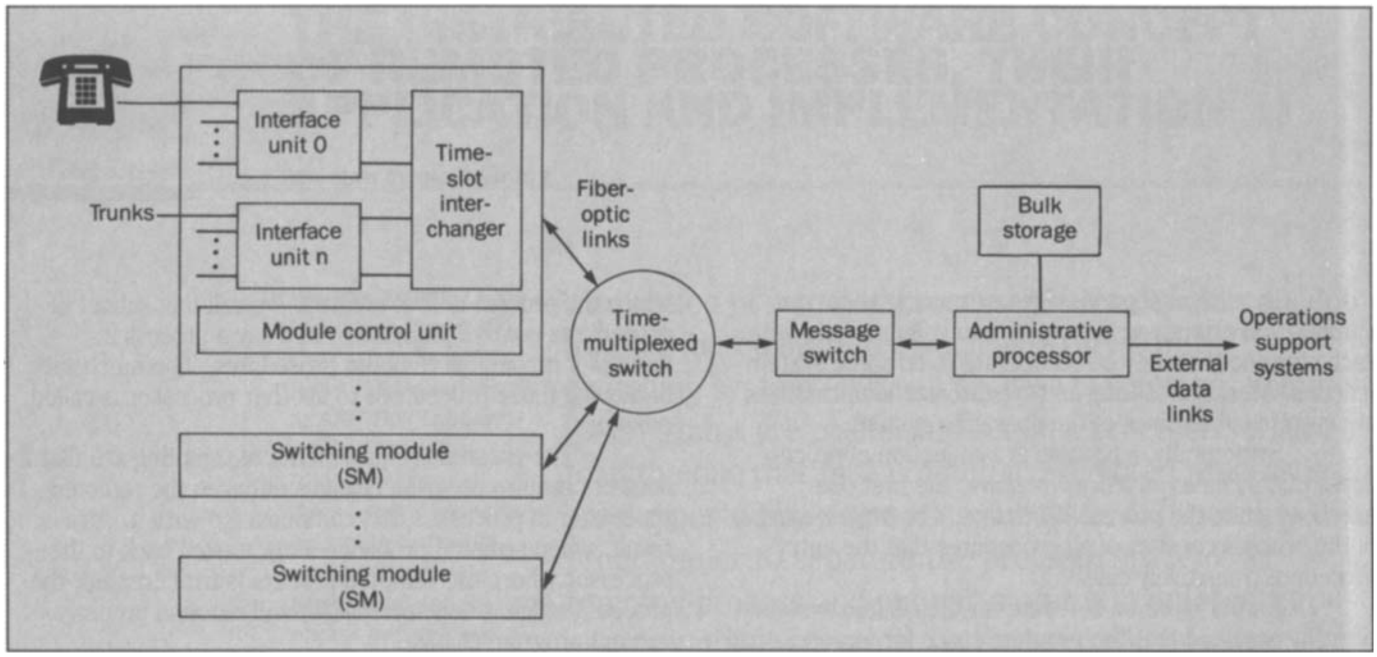

Figure 1. 5ESS™ switch system architecture. The switching modules directly control peripheral units for telephone lines, trunks, and other equipment. The data links carry control messages and provide communication paths for transmission lines in different switching modules.

niques to enhance reliability, some of our experiences with the implementation, and other related issues.

The Implementation Environment

As Figure 1 shows, the 5ESS switch consists of a set of switching modules, each a separate processor, and an administrative processor linked together by fiber-optic data links. The switch is partitioned into two stages with a centralized time-multiplexed switch, and a time-slot interchanger in each switching module.

Each switching module contains the software that provides basic call-processing capabilities and other real-time-intensive features. Other software controls and maintains all peripheral hardware in the switching module. The administrative processor handles routing and terminal allocation, and some call-processing features that are less real-time intensive or are used infrequently.

The remoting subsystem is used for the call-processing software. In our application, a remoted process is typically first created in a switching module, and its remoted procedures reside and run in the administrative processor. These remoted procedures and the programs that remain in the switching module all use the same operating system. Thus, the same kind of operating system services are available to programs in the different processors.

The operating system provides a set of message-passing primitives for processes to communicate with each other. Messages are tagged with the sender's process identifier and are addressed to the receiver's process identifier. To send a message to a message queue in another process or read one from its own queue, a process calls primitives—such as `send message` or `read message`—from its procedures.

The interface between remoted procedures and other procedures of a remoted process is by procedure call. Remoted procedures should be able to call other procedures across processor boundaries and vice versa. As a result, the remoting subsystem must be able to support *remoted procedure calls.*

Message passing provides the interface between a remoted process and other processes. Any procedure of the remoted process, even one placed in any processor, can interpret a message that another process sends. The remoting subsystem must then forward the message to the appropriate procedure to be interpreted.

Furthermore, although more than one processor may execute the procedures of a remoted process, the remoting subsystem must maintain a single process identification for the remoted process. It is the only identifier

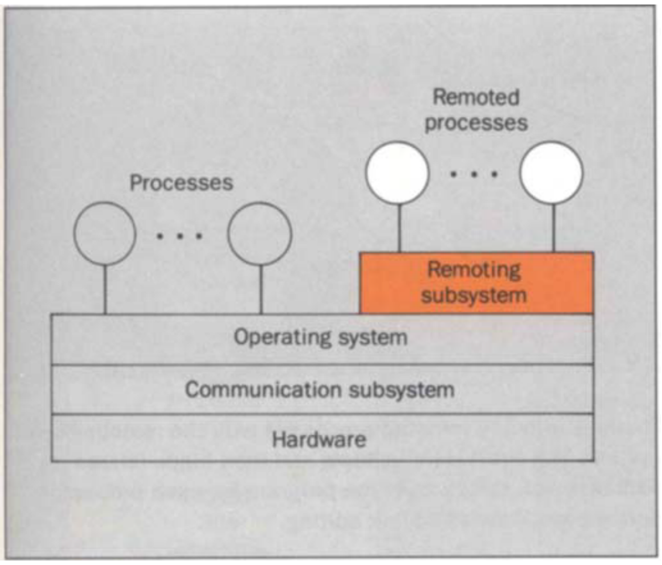

Figure 2. Virtual machines that support remoted processes. The remoting subsystem is a virtual machine built on top of the operating system.

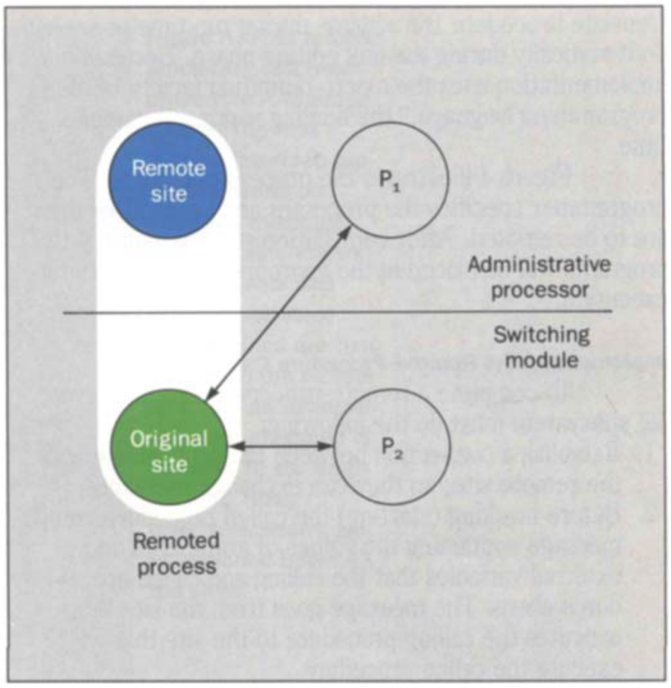

Figure 3. The two control sites or processes of a remoted process. The original site runs programs in the processor where the remoted process is first created, and the remote site runs the remoted procedures. Other processes (P1, P2) do not know about the remote site; their interface with the remoted process is preserved.

that the other processes will know.

The sections, "Implementing the Remote Procedure Call" and "Preserving the Communications Interface," describe the methods that we used to preserve these interfaces.

Running a Remote Process. A remoting subsystem is a virtual machine. In our implementation, it is built on top of the operating system (Figure 2) and is bound to the procedures of the remoted process during compile time. (Binding means the remoting sybsystem knows the address in memory for each procedure.)

Actually, as Figure 3 shows, two processes, called the *original site* and the *remote site*, execute the procedures of a remoted process. The original site runs procedures in the processor where the remoted process is first created, and the remote site runs the remoted procedures.

Because of the remoting subsystem, this separation is not visible to processes that communicate with the remoted process, nor is it a concern for the person who programs the remoted process.

Binding a Remote Process. One objective is that the remoting subsystem does not change the source code of the remoted process. When procedure A calls procedure B and procedure B is placed in another processor, clearly procedure A cannot be bound directly to procedure B. Instead, procedure A must be bound to some programs in the remoting subsystem.

We can do this binding in several ways without changing procedure A's source code. For instance, we can translate procedure B's address during run time or we can do it statically during the link editing phase. Because our implementation uses the macro-definition facility of the C programming language,[9] the binding is done at compile time.

Figure 4 illustrates the process of binding. The programmer specifies the processes and procedures that are to be remoted. After compilation and link editing, the programs will be placed in the appropriate processors for execution.

Implementing the Remote-Procedure Call

To complete a remote-procedure call, the remoting subsystem must do the following:

1. Establish a connection between the original site and the remote site, so they can exchange messages.
2. Before invoking (starting) the called procedure, send a message containing the values of arguments and external variables that the calling and called procedures share. The message goes from the site that executes the calling procedure to the site that will execute the called procedure.
3. After the called procedure is executed, send a message containing the values of returned variables and shared external variables. The message goes from the site that executed the called procedure to the site that executes the calling procedure.

The First Remote-Procedure Call. Our implementation required two engineering decisions: The remote site of a remoted process is not created until the first remote-procedure call is made. And, the remote site terminates after the return of the procedure that it executes first.

We made these decisions because, in our application (processing a telephone call), only those feature programs that control some specialized features, like call waiting, are remoted. Most telephone calls do not invoke these features. Furthermore, it is unlikely that a telephone call would involve more than one feature.

As a result, the first remote-procedure call sets up the remote site and implies the invocation of a call-processing feature.

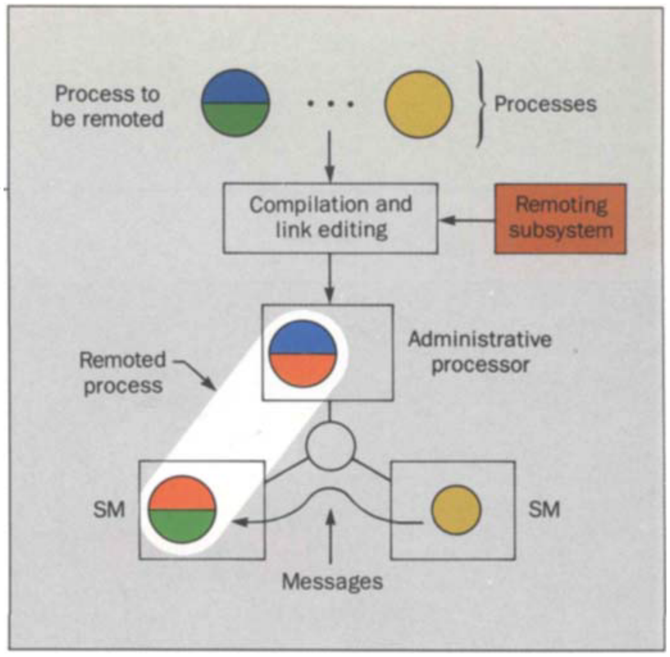

Figure 4. Binding remoted processes with the remoting subsystem. The subsystem collects and then binds (stores the addresses of) the parts of the program for each processor during compilation and link editing.

The first remote-procedure call involves a pair of procedures (called *feature interfaces*), a *remoting system* process, and a *remoting switch* procedure. The calling procedure is bound to the first feature-interface procedure rather than to the called procedure, which is in another processor. Therefore, the feature interface procedure is invoked instead of the called procedure.

The feature interface procedures serve two functions. They start the invocation of the called procedure, and they handle exceptions in case of failures.

When the first feature-interface procedure is executed, it sends an initiation message to the remoting system process in the other processor. This process, which is created when the system is initialized, does not terminate. When it receives the initiation message, the remoting system process creates the remote site and returns an acknowledgment message to the original site.

The second feature-interface procedure is the remote site's entry procedure. It calls the called procedure (after the remote site is set up) and passes it the values of arguments and external variables obtained from the initiation message. Figure 5 illustrates this sequence of actions.

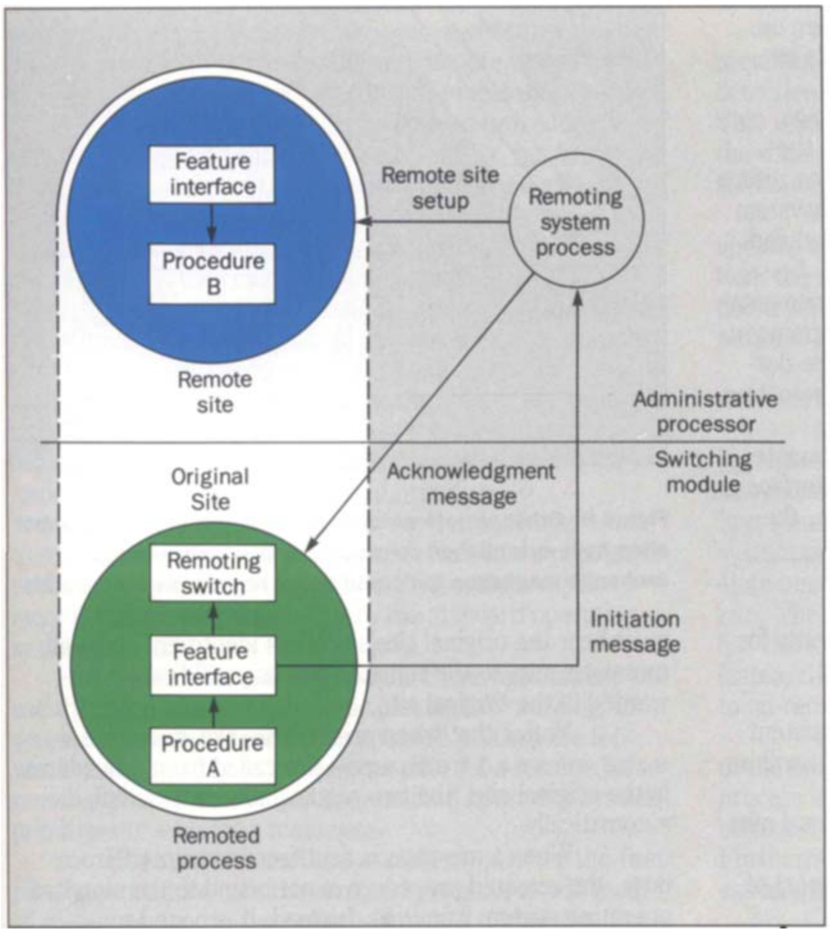

Figure 5. First remote-procedure call from procedure A to procedure B. The first feature-interface procedure sends an initiation message that contains values of arguments and external variables. These values are then passed to the remote site, and an acknowledgment message is returned to the original site. Now, communications are established between the sites.

Remoting Switch. After sending the initiation message, the original site of the remoted process enters a dormant state inside the remoting switch procedure. (A control site is in a dormant state if it is not running programs that belong to the remoted process.) The original site returns to an active state when the remote site initiates a remote procedure call to the original site or when the first remote-procedure call returns.

Besides responding to remote procedure calls, the remoting switch procedure maintains communications between the two control sites. It handles incoming messages to the remoted process and, when appropriate, forwards the messages to the remote site. If the communications link between the two processors fails, the remote switch also does much of the exception handling.

Other Remote-Procedure Calls. The acknowledgment message that the remoting system process sends to the original site contains the remote site's process identifier. The remoting system process also passes the original site's process identifier to the remote site; hence, the two control sites have established connection. From then on, the two control sites can do remote procedure calls by exchanging a pair of messages (Figure 6).

The first message, a procedure-call message, contains an index number to the called procedure and the values of the arguments and external variables. The return message contains the return values. Remoting subsystem programs, called remote-interface procedures, send and receive these messages.

Because a process specifies a sequential computation, when one control site is executing some programs of the remoted process, the other control site must be dormant. This site enters the dormant state inside a remoting switch procedure.

As we mentioned earlier, when the first remote-procedure call returns, it returns to the feature interface in the remote site. After sending the return message, the feature interface terminates the remote site.

Preserving the Communications Interface

As stated earlier, there are two requirements for preserving the interprocess interface of a remoted process:

1. The remoting subsystem must maintain a consistent process identifier for a remoted process, and it will be the only identifier known to other processes.
2. The remoting subsystem must be able to forward messages from other processes to the appropriate procedure (which may be placed in any processor) of the remoted process.

Consider the first requirement. In our application, a remote site is created only after the invocation of the first remote-procedure call. By then, however, the remoted process may have already established communications with other processes and those processes will already know the original site's process identifier.

To avoid affecting the way these processes communicate with each other, it follows that the remoted process should use the original site's process identifier. Consequently, all messages sent from the remoted process must bear the original site's process identifier. Thus, all messages sent to the remoted process will first be forwarded to the original site.

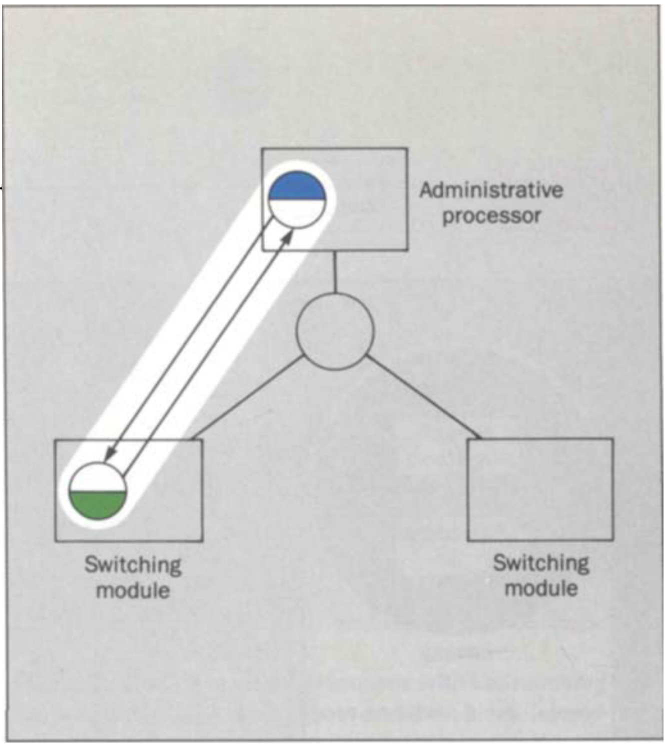

Figure 6. Other remote-procedure calls. Once the two control sites have established connections, they only need to exchange messages to do additional remote-prodecure calls.

Notice that when primitives—like `send message` and `read message`—are called from procedures in the original site, the two requirements are satisfied automatically.

When a message is sent from a remoted procedure, the remoted procedure is not bound to the standard operating-system primitive. Instead, it is bound to a remoting subsystem replacement that sends the message with the original site's process identifier. This satisfies requirement 1.

Message Forwarding. Our remaining problem is to decide when to forward messages from other processes to the remote site.

We implemented two algorithms: message forwarding on demand and automatic message forwarding. Programmers of a remoted process can choose to use either one, depending on the application's characteristics.

Message Forwarding on Demand. When message forwarding on demand is used, messages sent to the remoted process are not forwarded to the remote site unless that site has asked the original site for these messages.

To look for a message, a remoted procedure is bound to a remoting subsystem replacement, not to the standard operating-system primitive, `read message`. Because the replacement is executed first, it sends a request message to the original site, which, at that time, is dormant inside the remoting switch program.

After receiving the request message, the original site will send the first message in its queue to the remote site. If no message is found in the queue, a return message is sent to notify the remote site of the result. The replacement procedure, which will receive this message first, returns to the remoted procedure that called it with the same return value as the standard primitive.

Clearly, it will take more work and a longer time to read a message in the remote site. Because a process is executed sequentially, the remoting subsystem replacement is functionally equivalent to the standard operating-system primitive.

Automatic Message Forwarding. Some applications can use the more-efficient automatic message forwarding, in which all messages sent to the remoted process are forwarded immediately to the remote site. As a result, the remoted procedure can use the standard operating-system primitives to look for a message.

However, this method works only if, after the first remote-procedure call, the remoted procedures will read all messages sent to the remoted process. Otherwise, race conditions may occur.

We have provided one more method of optimization. A remoted process can switch from message forwarding on demand to automatic message forwarding if, from then on, all messages sent to the remoted process will be used only by the remoted procedures. Because of possible race conditions, a remoted process cannot switch from automatic message forwarding to message forwarding on demand.

Reliability Considerations

Call-processing applications have strict reliability requirements. As a result, much attention is paid to fault detection and fault recovery in call-processing software. This section briefly describes some techniques we used in the remoting subsystem for fault detection and some considerations for exception handling.

Fault Detection Techniques. The remoting subsystem updates a set of variables for a remoted process. In addition, the subsystem's auditing system process looks at these variables regularly to determine the sanity of the remoted process.

These variables are used in two different audits: a state variables audit and a message count audit.

State Variables. A process can have three remoting states. When first created, the process is in the *no-remoting state*. After the process has made the first remote-procedure call and received the remoting-system process's acknowledgment message, its state becomes *original*, to show that it is the original site. The remote site created in the other processor has the *remote* state. When the remote site terminates, the remoting state of the original site is reset to no-remoting.

Each control site also keeps the process identifier of the other control site. Thus, if the remoting state of a process is original (or remote), then we can identify the other control site, whose state will be remote (or original). Furthermore, the other control site will contain the process identifier of the first process.

The audit programs can use this relation to check the sanity of the remoted process.

Message Counts. For every message that a remoting subsystem program sends from one control site of a remoted process to the other, a message-sent count is updated by one. Similarly, for each message received from the other control site, a message-received count will be updated by one.

The following relations about the message count variables are true:

1. For the original site, the number of messages it sent is greater than or equal to the messages it received plus one. (The original site sends the first message to start off remoting.)
2. For the remote site, the number of messages it received is greater than or equal to the messages it sent plus one.
3. The number of messages sent from the control site that was examined first is less than or equal to the number of messages received by the other site used.

If messages are sent and received instantaneously, the messages sent from one site equals the messages received by the other site. However, after the audit program has examined the number of messages sent from one site, some time must have elapsed before the audit program can read the number of messages that the other site received. Hence, relation 3 is true.

These relations are checked periodically to determine if messages have been lost or if some messages have been misrouted to the remoted process.

Other techniques are used, besides setting up these variables, to check the remoted process's sanity.

One way is to use a timer in the first remote-procedure call. If the acknowledgment message is not received within a reasonable time, an error will be returned to the feature-interface procedure, which will then decide to retry or do other kinds of error handling.

Exception Handling Considerations. When a fault condition or an error is detected in a procedure, the procedure can either handle the error or report the exception condition to its calling procedure. In our implementation, the remoting subsystem does not do error handling. Instead, it reports the exception condition after the fault has been detected.

We chose this approach, because the call-processing programs are in a better position to determine error handling strategy. Depending on the call phase, the call-processing programs can decide to abandon the call, invoke terminal maintenance programs when errors occur, or take other appropriate actions. For the remoting subsystem to handle error conditions well, it must share substantial information with the call-processing programs, which requires that call-processing programmers participate in the design of the remoting subsystem.

During a remote procedure call, the remoting subsystem can encounter two types of exceptions:

- exceptions that the called procedure returned
- exceptions that occurred because of the remoting mechanism; e.g., when a *remote procedure call* message cannot be sent.

The remoting subsystem returns exceptions of the first type to the calling procedure without modification. It returns those of the second type as undifferentiated exception conditions whose reason or origin of error is not known. As a result, the error handling method of the calling procedure does not need to be changed for remoting.

Conclusions

We have described the essential features of a remoting subsystem for the 5ESS switch and discussed some reliability issues. Our implementation meets the stringent response-time requirements of call-processing applications. In addition, the subsystem meets the high reliability needs of the continuously running, call-processing environment. Some remoted processes in the first issue of the 5ESS switch software have since been unremoted and function properly without a program change.

Given a remoting subsystem, one can, in theory, choose to remote any subset of the procedures of a process. However, remote procedure calls and message forwarding to remoted procedures are not without extra expenses. In our case, we have judiciously chosen the programs to be remoted, so high-usage programs remain in the switching module. This is similar to the practice in virtual memory systems where high-usage programs remain in main memory.

Acknowledgments

The author would like to thank his colleagues who worked with him to realize the concept of remoted processes, especially T. S. Chow and J. W. Johnson.

References

1. S. M. Bauman, R. J. Carline, J. S. Nowak, and H. Oehring, "No. 5 ESS Software Design," *Proceedings of the 10th International Switching Symposium,* Montreal, September 1981, Section 31A.
2. P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM,* November 1978, pp. 934-940.
3. R. P. Cook, "*MOD—A Language for Distributed Programming," *IEEE Transactions on Software Engineering,* November 1980, pp. 563-571.
4. F. T. Andrews, Jr. and W. B. Smith, "No. 5 ESS—Overview," *Proceedings of the 10th International Switching Symposium,* Montreal, September 1981, Section 31A.
5. J. H. Davis, J. Janik, Jr., R. D. Royer, and B. J. Yokelson, "No. 5 ESS System Architecture," *Proceedings of the 10th International Switching Symposium,* Montreal, September 1981, Section 31A.
6. J. W. Johnson, J. C. Kennedy, and J. C. Warner, "No. 5 ESS—serving the present, serving the future," *Bell Laboratories Record,* Vol. 59, No. 10, December 1981, pp. 290-293.
7. B. J. Nelson, "Remote Procedure Call," Ph.D. dissertation, Report CMU-CS-81-119, Carnegie-Mellon University, Philadelphia, Pa., 1981.
8. A. Z. Spector, "Performing Remote Operations Efficiently on a Local Computer Network," *Communications of the ACM,* April 1982, pp. 246-260.
9. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, N. J., 1978, pp. 86-87, 207.

(Manuscript received December 23, 1985)