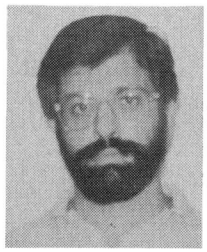
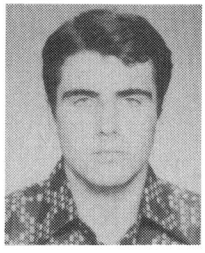
retrieval using indexed descriptor files," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 522–528, Sept. 1980.
[13] C. S. Roberts, "Partial match retrieval via superimposed codes," *Proc. IEEE*, vol. 67, pp. 1624–1642, Dec. 1979.
[14] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval (Comput. Sci. Series)*. New York: McGraw-Hill, 1983, ch. 3, pp. 52–99, ch. 4, pp. 118–151.
[15] D. G. Severance and G. M. Lohman, "Differential files: Their applications to the maintenance of large databases," *ACM Trans. Database Syst.*, vol. 1, pp. 256–267, Sept. 1976.
[16] B. Shneiderman and V. Goodman, "Batched searching of sequential files and tree structured files," *ACM Trans. Database Syst.*, vol. 1, pp. 268–275, Sept. 1976.
[17] S. Stiassny, "Mathematical analysis of various superimposed coding methods," *Amer. Documentation*, vol. 11, pp. 155–169, Apr. 1960.
[18] M Taube, A. Kreithan, and L. B. Heilprin, "Superimposed coding for data storage with an Appendix of dropping fraction tables," in *Mechanization of Data Retrieval (Studies in Coordinate Indexing Series)*, vol. 4. Washington, DC: Documentation, Inc., 1957, ch. 5.
[19] D. Tsichritzis, "Form management," *Commun. Ass. Comput. Mach.*, vol. 25, pp. 453–478, July 1982.
[20] D. Tsichritzis and S. Christodoulakis, "Message files," *ACM Trans. Office Inform. Syst.*, vol. 1, pp. 88–98, Jan. 1983.
[21] D. Tsichritzis, S. Christodoulakis, P. Economopoulos, C. Faloutsos, A. Lee, D. Lee, J. Vandenbroek, and C. Woo, "A multimedia office filing system," in *Proc. VLDB*, Florence, Italy, 1983, to be published.



Stavros Christodoulakis received the B.Sc. degree in physics from the National University of Greece, Athens, Greece, in 1971, the M.Sc. degree in computer and information sciences from Queen's University, Kingston, Ont., Canada, in 1977, and the Ph.D. degree in computer science from the University of Toronto, Toronto, Toronto, Ont., Canada, in 1981.

He is currently an Assistant Professor in the Department of Computer Science, University of Toronto. His research interests include office information systems, extended database management systems, information retrieval, integration of text, image and voice data, performance evaluation, and pattern recognition.



Chris Faloutsos (S'79) received the B.Sc. degree in electrical engineering from the National Technical University of Athens, Athens, Greece, in 1981 and the M.Sc. degree in computer science from the University of Toronto, Toronto, Canada, in 1982.

He is currently working towards the Ph.D. degree at the University of Toronto. His research interests include information retrieval, office automation, and performance evaluation.

Mr. Faloutsos is a member of the Technical Chamber of Greece.

Debugging a Distributed Computing System

HECTOR GARCIA-MOLINA, FRANK GERMANO, JR., MEMBER, IEEE, AND WALTER H. KOHLER, MEMBER, IEEE

Abstract—**In this paper we discuss the issues involved in debugging a distributed computing system. We describe the major differences between debugging a distributed system and debugging a sequential program. We suggest a methodology for distributed debugging, and we propose various tools or aids.**

Index Terms—**Bottom-up debugging, debugging, distributed computing system, monitoring, tracing, two-phase debugging.**

I. Introduction

A *distributed computing system* is a collection of communicating and cooperating processors which are working towards a joint goal [10]. For example, the goal may be to provide the user with a database management system or with an office information system.

The term "distributed computing system" can refer to the system hardware, the system software, or both. The hardware includes the processors or nodes, the processor interfaces to the communications subsystem, and the communications subsystem itself. The software is the collection of programs which runs on the processors. These programs operate in an integrated fashion in order to achieve the common system goal. In this paper, we will use the term "distributed computing system" to refer to the software only.

This *distributed software*, like all software written, is prone to errors or bugs [22]. Given that we have evidence of a bug, *debugging* is the process of diagnosing the cause of the problem and correcting the software to remove it. In this paper we will concentrate on the diagnosis aspect of debugging since we consider the correction of the software (after the bug has been identified) to be relatively straightforward.

The debugging of a distributed computing system, which we will address here, is in essence similar to the debugging of sequential computer programs [4], [17], [19], [27]. All

Manuscript received August 2, 1981; revised July 26, 1983. This work was supported in part by the National Science Foundation under Grant ECS-8019393 and by the Digital Equipment Corporation.

H. Garcia-Molina is with the Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ 08540.

F. Germano, Jr. is with Apollo Computers, Inc., Chelmsford, MA 01824.

W. H. Kohler is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01002 and with the Digital Equipment Corporation, Hudson, MA 01749.

0098-5589/84/0300-0210$01.00

computer programmers are familiar with *sequential program debugging:* after a program is written, various test cases are run (all on a single computer). When unexpected behavior is observed, the program, its behavior, and its output are analyzed in order to discover the source of the problem.

Debugging sequential programs is an art. It requires patience, attention to detail, and is time-consuming. To aid the programmer, some facilities and tools are usually available. These tools include interactive symbolic debuggers, interpreters, memory dumps, audit and trace packages, and others [19]. As the sequential program grows in size and complexity, good debugging tools become imperative.

Of course, it can be argued that it would be better to try to avoid bugs when the program is written. Structured programming [7] and program verification [13], [18], [20], etc. are useful techniques for reducing the number of bugs in programs. However, even with these techniques, bugs will exist and there will always be a need for debugging. Although we recognize the importance of good program design and of program verification (both in centralized and distributed environments), in this paper we will concentrate exclusively on debugging.

We have stated that in essence, debugging a distributed computing system is similar to debugging a sequential program. However, there are some important differences which make *distributed system debugging* an even more challenging task. These differences can be roughly grouped into four categories.

1) *Multiple Asynchronous Processes:* The distributed computing system is made up of a collection of processes which communicate via shared variables or via messages. (For the moment, assume that all the processes are on a single computer. Later on we discuss the complications that arise because the processes may actually be executing on separate processors.) The fact that the system has various loci of control makes it harder to understand and more prone to bugs. The bugs in many cases are subtle or sporadic, caused by improper synchronization among the processes or by race conditions. As an added complication, the results observed in the system are in many cases hard to reproduce [3], [26]. This is because the results do not only depend on the system input, but also depend on the relative timing of the processes.

2) *Multiple Processors:* The processes that make up the distributed computing system can be running on different physical processors. Having multiple processors makes it difficult to discover erroneous program behavior and to manage the debugging process. To illustrate this, assume temporarily that all the processes run on a single processor in a time shared fashion. If one of the processes executes an illegal operation (e.g., divide by zero), then it is relatively simple to halt all processes (after all, only the process performing the illegal operation was active). The programmer can immediately tell what process was at fault and what the error was. (Of course, finding the cause of the error will probably take some time.) Now consider a similar situation where the processes run on different processors. The processor executing the illegal operation can be halted immediately, but other processes will continue running. In order not to lose any critical information regarding the bug, the other processes should be stopped "as soon as possible," but this will still take some time. By the time all processes are stopped, the critical information may have been destroyed and it will become very difficult to discover the cause of the problem. Furthermore, when the distributed system does stop, the programmer will have a harder time just discovering what process failed and what the immediate cause of the failure was. (That is, with one computer the programmer just has to look at one "program counter" and one "error status word" in order to know what process failed and why. Now the programmer must examine the states of multiple computers.)

Although in this paper we are not considering hardware failures, it may be worth noting here that hardware failures may cause considerable grief to the programmer debugging software running on multiple processors. If the entire system runs on a single computer, many hardware (or even operating system) failures will be obvious to the programmer. On the other hand, with multiple processors the programmer may not be aware that one of the processors is down and causing the system to behave in an erroneous way.

3) *Significant Communications Delays:* The processors on which the distributed computing system is running can be geographically dispersed. This may introduce significant communications delays, which can in turn, make some debugging operations impractical. For example, consider a process which stores on disk a lengthy trace of its operations. In a single computer environment, it may be possible to examine this trace through a standard text editor. However, if the trace file and the person examining it are on geographically separated computers connected with slow communications lines, then the examination of the trace file becomes difficult: both transferring the entire file or performing remote text editing may be too slow.

4) *Size of the System:* Distributed computing systems have a tendency to be "large," that is, to have a large number of processes and processors. For example, distributed office information systems can have hundreds of intelligent user work stations; distributed banking systems can have computers at tens of branch offices. As we know, the larger a system is, the harder it is to debug. Of course, "largeness" is not an exclusive property of distributed computing systems. But, in general, we can expect the debugging of a distributed computing system to be more arduous and time-consuming than the debugging of a sequential program.

As one would expect, the techniques and tools that have been developed for sequential program debugging are simply not adequate for distributed system debugging. Typically, distributed computing systems are currently debugged as follows: each process in the system is run through a standard interactive debugger. To avoid confusion, the output of each debugger is displayed on a separate terminal. This not only requires many terminals connected to various computers, but also requires a lot of moving (or running) between terminals! The moving could be eliminated by using a "smart terminal" which somehow multiplexes all the terminals into a single one. (For example, the output of each terminal could be viewed on a "window" of the smart terminal.) However, this does not really solve the hard problems. It has been our experience,

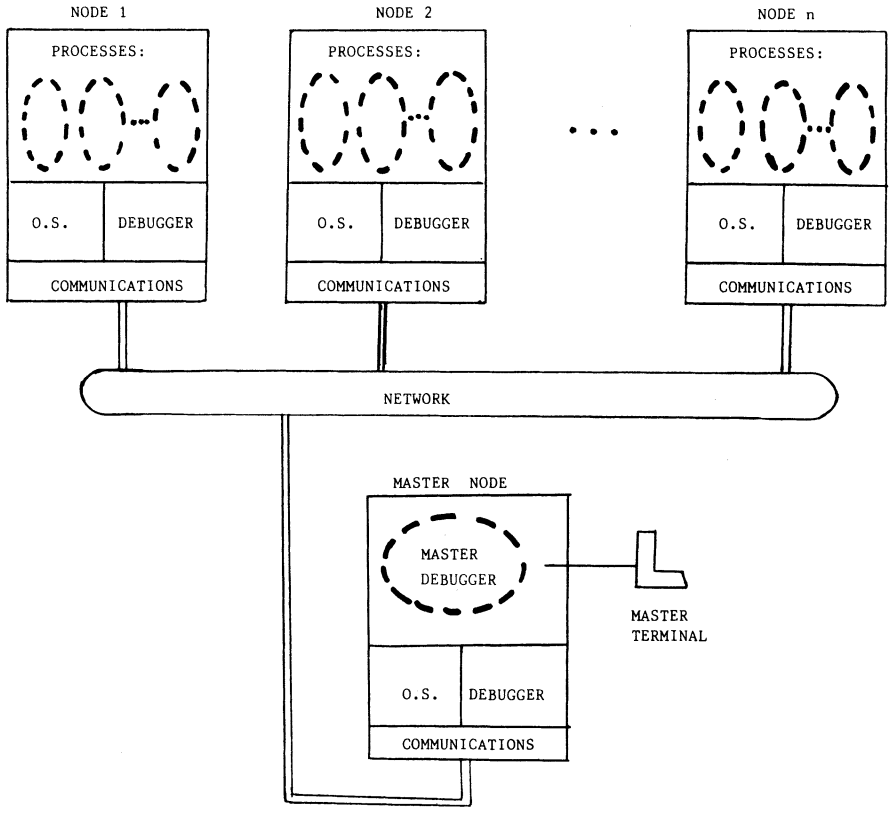


Fig. 1. The model.

and that of other researchers, that something better is needed. We need new methodologies and tools which are specifically designed for distributed computing systems. It is the objective of this paper to make some suggestions in this direction.

In Section II, we define our model of a distributed computing system. Then in Section III, we discuss a methodology for distributed system debugging. In Section IV, we suggest some debugging tools. Finally, in Section V, we make some concluding remarks and briefly describe our experience in using some of the techniques and tools described.

We wish to emphasize that this paper does not describe a fully implemented and complete distributed debugging system. To our knowledge, no such system currently exists. The authors have implemented and debugged a distributed computing system, and wish to share this experience by proposing a methodology and tools for distributed debugging. Had we had these aids when we implemented our system, our debugging task would have been simplified tremendously. And even if in the near future these tools are not available to implementors, we feel they can be built into a specific application as it is developed. As a matter of fact, we built in some of these tools into our application as we went along and found them to be very useful. We will discuss our experience in more detail in the conclusions section.

II. The Model

When a high level symbolic debugger for sequential programs is described, no mention is made of how compilers, debuggers and other low level code is debugged. Here we will use a similar strategy. We are going to assume that the necessary low level software is debugged, and we will concentrate on the process of debugging the high level or applications code. The debugged low level software includes the operating system running on each processor, the communications subsystem, and the debugging tools. The software to be debugged is the one running on top of the operating system, communications subsystem, and debugger. Typical application software in this context includes a distributed database management system, a distributed transaction processing system, a distributed office information system, an electronic mail system, and others.

We view the distributed computing system as running on a collection of processors or computers. The processes which make up the application system, called the application processes, are distributed on these computers. In addition to this code, each computer has the low level code which supports the application processes. We view this code as being divided into three modules: the operating system module, the communications module and the debugging module. Fig. 1 illustrates this in a very simplified fashion. (In our model, we ignore all code and processes which are not involved in the distributed system being debugged. Also note that some of the debugging module code may actually be located inside the application processes when they are being debugged.)

The operating system module is in charge of scheduling the application processes and managing the local resources. The communications module is responsible for intercomputer communication. The debugging module contains the code used for debugging the local processes. It is important to note that the debugging module at a node does *not* simply contain the standard sequential program debugging tools. Each debugging module must be viewed as a component

of a distributed debugging facility. That is, the collection of debugging modules at the computers, in a coordinated and integrated fashion, provide the desired debugging facilities.

We will use the term *programmer* to refer to the person who is debugging the distributed system. The programmer must be logged on to one of the computers in the system. For simplicity, we assume that the programmer is on a computer which has no application processes. This computer, which we call the *master* computer, only runs a process called the *master debugger.* The programmer, through the master debugger, controls the debugging of the distributed system. The input/output device through which the programmer and the master debugger communicate is called the *master terminal.*

Processes can communicate via shared variables or via messages. (Of course, to communicate via shared variables, processes must have access to common memory.) Accesses to shared variables are handled by the operating system module, which is in charge of synchronizing these accesses. Message passing is also done through the operating system. When the operating system observes a message destined for a remote computer, it hands the message to the communications module.

The debugging facilities must be carefully integrated with the operating system, the communications subsystem and with the programming language(s) in use. For example, if necessary, the debugging module at a node must be able to monitor all interprocess communications. The debugging modules may have to communicate among themselves via the communications module. The debugging module must be able to tell where processes keep their local variables and what the names of these variables are.

Note that our point of view is that the code is not being tested on a simulator [23] but is running on the actual hardware. Nevertheless, many of the comments we will make still apply to debugging with a simulator and most of the tools we will describe can be integrated with a simulator.

Finally, we must point out that *any* debugging facility may interfere with the operation of the system, and may either cause new bugs to appear or may make it impossible to observe existing bugs. For instance, the processing delays caused by the debugging itself may make certain subtle synchronization errors appear or disappear. The new bugs may still be studied, but obviously, the "unobservable" ones will be difficult to correct. (Their effects will still be observable when the debugging facilities are not in use.) Since there is very little that can be done to avoid these problems, we will not discuss them further. However, they should be kept in mind by the programmer.

III. Distributed System Debugging

Let us picture ourselves in charge of a large and complex distributed computing system. How shall we proceed in order to find the bugs in the system with the least effort? What strategy or methodology will give best results? The answers to these and similar questions obviously depend on the specific system involved. Nevertheless, we believe we can make a few general observations on the debugging process. In the following three subsections we will argue that:

1) debugging a distributed system should usually proceed in a bottom-up fashion;

2) debugging will mostly be based on the judicious use of traces; and

3) locating a specific bug will typically be done in two phases.

A. Bottom-Up Debugging

As described by Lauesen, "In bottom-up debugging, each program module [or process in our terminology] is tested separately in special test surroundings. Later, the modules are put together and tested as a whole. In top-down debugging, the entire program is always tested as a whole in nearly the final form." (The test program may have extra output statements and some sections may be replaced by dummy sections [17].)

Each of these techniques has its own set of advantages and disadvantages. When debugging a sequential program, a top-down approach can eliminate the need for artificial test surroundings and thus save programming effort [17]. However, in a distributed system the effort required to find a bug can be so great that it becomes worthwhile to invest the time and effort to create the artificial test surroundings. Another potential advantage of top-down debugging is that "misunderstandings between communicating modules are revealed early in the debugging process" [17]. This is certainly true, and in a distributed environment it is important to test and debug the interfaces among the processes (or modules) as early as possible. Still, this checking will be facilitated if the processes have been at least "superficially debugged." That is, due to the usual complexity of distributed systems, finding the process interface bugs will be difficult unless the processes being checked are working properly.

The argument for bottom-up debugging can be worded as follows: in a distributed program, the road to a "bug free" system is more hazardous than in a sequential program. Thus, the programmer must proceed cautiously and slowly. The programmer must be conservative. One way to be conservative is to use bottom-up debugging.

In order to test and debug a single process, we do not really need distributed debugging tools. The standard sequential program debugging tools will usually be adequate. This is true even if a few auxiliary test processes have to be used in order to drive the process we are checking. The standard tools are satisfactory because all the processes will probably run on a single computer (eliminating the problem of multiple processors and communication delays discussed in the introduction). The auxiliary processes should be simple, making it reasonable to assume that they operate correctly. Furthermore, the interaction between the main and auxiliary processes should also be simple (this avoids subtle synchronization errors), and the entire "system" should be relatively small.

Typically, a single process is tested and debugged as follows. The programmer starts up the auxiliary processes, if any. Then the main process, with an interactive symbolic debugger compiled in, is run. The programmer, through the interactive debugger, controls the execution of the main process. The programmer can set break points and examine the process var-

iables in order to discover any bugs. (Notice that the programmer should not have to examine or control the auxiliary processes since they are not doing much.)

After processes are individually tested, they can be combined and tested using the distributed debugging tools we will soon describe. Hopefully, after having debugged processes individually, most of the bugs that remain will be the really challenging ones, i.e., the ones resulting from the interaction between the processes. The distributed debugging tools are intended for locating these bugs. But notice that the distributed tools do not replace the sequential program tools. On the contrary, both types of tools should be used in unison for debugging distributed computing systems.

B. Traces

As mentioned in the previous subsection, the distributed debugging tools should be helpful in discovering bugs which result from the interaction of the various application processes. In many cases these bugs will only occur when the processes exchange data or synchronize their activity in a certain fashion or order. Since the observed results are hard to reproduce, we believe that tracing will play a prominent role in distributed system debugging.

A process trace records the history of the process [1], [3], [6], [24]. An entry is made on the trace for each "important" event that occurs in the life of the process. For example, the fact that a variable is changing values can be recorded in an entry. The branch taken in an IF statement may also be recorded. If each process in the distributed system keeps a trace, then the programmer has a written record describing what occurred in the system. This record can be examined by the programmer after a bug is observed in order to discover what it is that went wrong.

C. Two-Phase Debugging

Tracing process execution has a serious drawback: keeping the traces is time-consuming and requires large amounts of storage. In a distributed system, the problem is compounded by the fact that there are typically many processes. Furthermore, examining and analyzing such voluminous amounts of data becomes a problem in itself.

Thus, on the one hand, traces are in many cases the only way to discover subtle synchronization bugs which occur in distributed systems. On the other hand, in a distributed system traces can be prohibitively expensive to generate and use.

One possible way out of this dilemma is to significantly reduce the amount of data that is recorded in the process traces. This can be done by only making entries for the truly "significant" or "important" process events. It is difficult to precisely define which are the important events, since this depends on what the process is doing. However, the important events should include all events which directly or indirectly can affect other processes. On the other hand, all variable state changes and the outcome of all IF statements are not necessarily important events. Stated another way, the trace should be "high level," ignoring most low level details. (In Section IV we will discuss how these traces can be produced.)

The key idea in reducing the amount of data stored in the process traces is the following one. By examining the combined process traces, the programmer should be able to discover the erroneous process and the conditions which caused the bug to occur. But it is not necessary that the programmer be able to accurately pinpoint the cause of the bug (say to the exact line of code) solely by examining the traces. In other words, we envision the debugging process as proceeding in *two phases*. In the first phase, the programmer runs the system until a bug is observed. The programmer then examines the traces produced by the run and identifies the process interaction that produced the error. In some cases the programmer may be able to define precisely the cause of the error just through the information in the traces. But in other cases, the programmer will have to proceed to a second phase where the erroneous process (or processes) are tested in an artificial environment which attempts to recreate the conditions under which the original bug was observed. Various tests may now have to be run before the exact problem is found. In this phase, the trace facilities may again be used, or the programmer may use the sequential program tools which are available.

Creating the artificial environment for the second phase may be a difficult and time-consuming task, especially if the bug is a synchronization one. So we emphasize that the second phase is not introduced to simplify the debugging process. It is needed because in many cases it is simply not feasible to collect all the necessary information during the first phase.

We should mention that the size of the trace files can also be reduced by only keeping the more recent data. That is, the file can be written in a circular fashion, the newest data being written over the oldest data. This is certainly a very useful technique, but does not totally eliminate the need for high level traces. As discussed earlier, with a high level trace the programmer is not swamped with many details which can obscure the interaction between processes. Furthermore, high level traces are less time-consuming, both in the time needed for recording them and in the time needed for examining them. (If it can be afforded, one option is to record both the high level and the low level information. This may eliminate the need for the second phase.)

IV. Tools

After having made some general remarks on the debugging process in the distributed environment, in this section we will now suggest some tools which we believe will be useful for debugging distributed computing systems. Section IV-A describes tools for producing process traces; while Section IV-B presents tools for examining these traces. As discussed in Section III, we consider these tools to be the most important ones for distributed systems. As complements to the tracing tools, in Section IV-C we describe a strategy for controlled system execution and in Section IV-D a facility for monitoring the system performance.

A. Tools for Producing Traces

In the first phase debugging of a distributed system, each process produces a *trace* which records its history. Then the

process start (input parameters, · · ·)
process end (output parameters, variable dump, · · ·)
message send (source, destination, message type, parameters, · · ·)
message receive (source, destination, message type, parameters · · ·)
resource lock (resource id, lock type, · · ·)
resource unlock (resource id, lock type, · · ·)
transaction begin (input parameters, · · ·)
transaction end (· · ·)
transaction commit (· · ·)
transaction abort (· · ·)
start trace (date, time, node, · · ·)
end trace (date, time, node, reason, · · ·)
time adjustment (amount, on message from, · · ·)
deadlock detected (participants, victims, · · ·)
directory lookup (requesting process, object, address, · · ·)
security authorization (user, type, granted?, · · ·)
file open (file name, mode, · · ·)
file close (file name, · · ·)
file access (file name, read/write, block, · · ·)
node failure/repair (node id, type, detecting node, · · ·)
communication failure/repair (line id, type, detecting node, · · ·)
enter critical region (process id, region id, · · ·)
P/V (· · ·)
user defined (· · ·)

Fig. 2. Important distributed system events.
(Event independent parameters not shown.)

programmer examines the traces in order to understand how and why the bug occurred. There are two types of tools that would be of use for this type of debugging: facilities for producing the traces and facilities for examining them. In this subsection we look at the first type of tool, leaving the second type for the following subsection.

One way of producing traces would be to require each person who writes code to add write statements at appropriate places. The output of these write statements would go directly to the trace file. Since the person who writes the code knows what the "significant" or "important" events in the life of a process are, the trace would be as succinct and useful as possible. As Knuth states, "The most effective debugging techniques seem to be those which are designed and built into the program itself–many of today's best programmers will devote nearly half of their programs to facilitating the debugging process on the other half" [14].

This is conceptually the way we think traces should be constructed in a distributed system. However, to simplify the work of the programmer, we propose that the operating system module and the debugger module at a node (see Section II) aid by automatically writing many of the trace file entries.

In a distributed system, there are a set of commonly occurring events which are critical to the understanding of the system operation. For example, the event of sending a message is a very common event which should be recorded in the process trace. The termination of a process is another example of a standard and important system event. The debugger module can record these events and relieve the programmer of this burden.

Fig. 2 gives a collection of important and common distributed system events. Some of these events require a further explanation, but we will first make some general comments with respect to this figure. For each of these events, the trace file will contain some standard parameters and some event specific parameters. Some of the more obvious event specific parameters are given in the figure. The standard parameters for each entry are as follows.

1) *Type of Entry:* This parameter defines the type of event (e.g., message receive). As shown in Fig. 2, one of the event types is "user defined." This type is used by the programmer when explicitly recording events which are not handled by the debugger.

2) *Process Identification:* In many cases it will be convenient to store all the process traces for a given node in a single file. Since the entries for the various processes may be interleaved, it is necessary to identify the process performing the event.

3) *Timestamp:* The timestamp is a number which uniquely identifies the time when the event occurred [16]. The timestamps provide a total ordering of the system events, and can be generated by logical clocks [16]. (Readers not familiar with logical clocks can simply assume that each computer has a perfect physical clock.)

4) *Transaction (or Job) Identification:* The concept of a *transaction* is a powerful one in many distributed systems. A transaction is a collection of related actions which are managed by the system as a unit. In a distributed database, a transaction can be a user request to increase the salary of all employees by 10 percent. In a distributed office information system, all operations performed to process a request for more widgets can be a transaction. A job to print a file at a remote location is also a transaction. Whenever an event being traced is caused by or is on behalf of a certain transaction, the identification of the transaction is recorded in the entry for the event. (Typically, a transaction identification consists of a name or type followed by the time and location of origin.)

We also want to point out that in some cases, the debugger (and operating system) will be unable to record the occurrence of all the evemts listed in Fig. 2. For example, if the operating system does not do deadlock detection, and instead the application processes perform this function, then it is impossible that deadlock detection events be recorded automatically. In this case, it will be up to the deadlock detection code to report when it has detected and broken a cycle.

In other cases, the events of Fig. 2 may not be "important" in a given application. For example, if a system has no security checks, then the event "security authorization" is irrelevant. Thus, the events should only be taken as representative examples.

We now explain the Fig. 2 entries which may not be obvious to some readers. To identify a trace file, the beginning (and ending) dates and times are recorded as events. These are called the begin trace and end trace entries.

To record the important events in the life of a transaction, we have the transaction begin, end, commit and abort events. The commit point of a transaction occurs when the system guarantees that the transaction will eventually complete. The abort point occurs when the system decides to roll back a transaction [12].

When nodes keep (imperfect) physical clocks, the clocks which lag behind must periodically be advanced [16]. Such time correction is recorded by the time adjust event.

As discussed in Section II, application processes can also

communicate via shared memory. Each interaction via shared memory should be recorded in the trace. For example, event "enter critical region" indicates that a process has entered its critical region (sometimes called a monitor) [2]. Events P and V record the execution of these synchronization operations [9].

The process end event records the termination of a process. If the termination is abnormal, a dump of the process variables can be added to the trace entry. This dump can be produced in the same way dumps for sequential programs are produced.

The trace files at a given node are managed by the debugging module at the node (see Fig. 1). This module is informed by the operatimg system and communications modules whenever an important event occurs, and the appropriate trace entry is made. An application process can also inform the debugging module of a user defined special event. This event is recorded on the trace file using exactly the same format as for the other entries.

Usually, the programmer testing and debugging the system wishes to have some control of the tracing process. Thus, the master debugger (see Fig. 1) should be able to instruct all debugging modules to start or stop collecting traces. Another useful feature would be for the traces to be selective. That is, the master debugger could tell the debugging modules to ignore certain types of events.

When using the tracing facilities, a test would proceed as follows. The application processes are stated at all nodes. To start the tracing, the master debugger sends out messages to all computers telling them to start saving the trace data. When the programmer observes any unexpected system behavior, he or she halts the tracing and execution by having the master debugger send out "stop" messages to all computers. So that the programmer can be aware of the unexpected events, all computers should keep him or her posted with respect to unexpected behavior. That is, whenever an unexpected event occurs at a computer (e.g., a process addresses nonexistent memory), a message with the appropriate information should be sent to the master debugger. This information will then be displayed on the master terminal, and the programmer will decide if the traces should be continued or not. (In Section IV-D we suggest another strategy for keeping the programmer informed of the system status.)

An alternative to having the programmer manually stop the traces and system execution, is to have the computer which encounters the unexpected behavior immediately send "panic" messages to all computers (including itself). When a computer receives a "panic" message, it records it in the trace file and then terminates all processes. (This also terminates all tracing.) This approach has two advantages. First, the programmer is relieved of the burden of supervising the operation of the system. Second, since the system is halted faster, it reduces the probability of having one error cascade and cause errors in other processes. The superfluous errors can easily confuse the programmer and make it harder to identify the original abnormal event.

B. Tools for Examining Traces

After a test run, we are left with a collection of trace files located at the various computers. These files actually constitute a distributed database. Therefore, to examine the trace data we can rely on the techniques that have been developed for managing and querying a distributed database [21]. Fortunately, we do not need the full power of a distributed database management system, since our data have a simple structure and we only wish to read it. In this section we briefly outline some of the ideas that may be useful for our debugging data.

The trace data can be viewed as a single relation (or table) [5]. The attributes (or column headings) of the relation are node id, event type, process id, timestamp, transaction id, and event dependent data. Each tuple in the relation (or line in the table) is a trace entry. Note that for convenience, we are grouping all the event dependent data into the last field. (The relation is thus not normalized. This is not the only way to view the data.) The relation is partitioned among the computers. That is, some of the tuples are located at one computer, other tuples at a second computer, and so on.

The advantage of viewing the data as a relation is that we can use relational languages to examine the data [8], [25]. This makes it relatively easy for the programmer to retrieve the entries he or she is interested in. For example, if the programmer is interested in knowing what messages were sent and received by process 352 after 10:00 AM, he or she could issue the following SEQUEL query:

```
SELECT event-type, timestamp, transaction-id, event-dependent-data
FROM   R
WHERE  (event-type = 'message send' or
        event-type = 'message receive') and
        process-id = 352 and
        timestamp > 10:00 AM
```

(R is the name of the relation with the debugging data. Of course, SEQUEL is just one of the many languages that could be used.) To make sense, entries should be displayed in either increasing or decreasing timestamp order. Other sample queries that could be posed are: "Display entries for transaction 53 in decreasing time order"; "Display all entries of type transaction-abort at nodes 1, 3, 7"; etc.

It would also be useful to be able to query the event dependent data. String matching facilities can be added to the relational query language for this. For example, assume that all the event dependent parameters are preceded by their name in the trace files. Suppose that we wish to examine trace entries where the resource 'line printer' was locked. Then we would search for the string "resource-id = 'line printer' " in the event dependent data of all entries of type resource lock.

As was mentioned in the previous subsection, the process end entries may contain a dump of the variables of the process. Assuming that the dump has some standard format, the string matching facilities can be used to investigate the state of the process when it stopped. (For example, we can look for the string "COUNTER = " to find the last value of variable COUNTER.)

The facilities for querying the trace database should be accessible to the programmer through the master debugger. To answer the queries, the master debugger must select an access strategy which is efficient. At one extreme, the master debugger could initially transfer all the trace files to the master computer, and then answer all queries from the local files. At the

other extreme, the master debugger can collect the data as it is needed. Each time a query is posed, the master debugger would distribute copies of the query to all computers. Each computer would retrieve the relevant entries and send them to the master debugger. Selecting the best strategy seems to be an open research question [21].

C. Controlled Execution

Traces may be very useful, but in some special cases we may want to run the system in a controlled fashion, observing the intermediate states. This is analogous to running a sequential program one instruction at a time under the control of an interactive debugger. Or it could also be analogous to running the program until a breakpoint is encountered.

If we are to use these ideas in a distributed system, we must be careful. Since they are multiple processors, what do we mean by "one instruction at a time?" Out of the various candidates, which is the "next" instruction to be executed? If we set a breakpoint in a process, do we just want to stop that process? Or do we want to stop the entire system? And how is the entire system stopped?

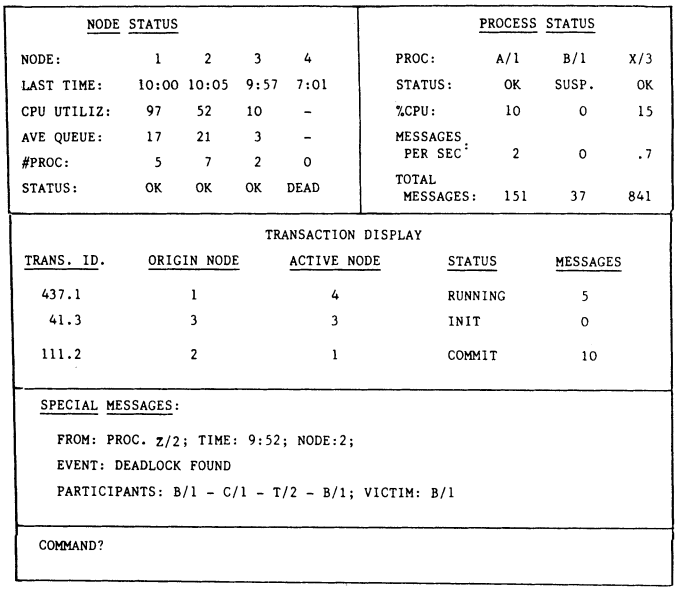
These questions may be answered in different ways, and in this subsection we will only suggest one set of solutions. First we discuss how the system can be temporarily stopped. Then we describe how the current state of the system can be examined by the programmer. Finally, we look at the issues of executing one instruction at a time.

We believe that in order to properly examine the state of the system, all processes should be stopped. This can be achieved by broadcasting "pause" messages to all computers. A pause message is similar to the panic message discussed earlier, except that now processes are only temporarily suspended. The broadcast of the pause message is triggered either by the programmer or by a process which hits a breakpoint. (The breakpoint can be set in a process in the same way that breakpoints are set in sequential programs.)

Once all processes are suspended, the programmer, through the master debugger, should be able to examine the state of the processes. This is straightforward if we assume that the debugging module at each node contains an interactive debugger. Thus, if the programmer wishes to know the contents of variable COUNTER of process 53, a request is routed to the computer running process 53. At that computer, the interactive debugger is used to look into the storage area of process 53, and the requested value is forwarded to the master debugger. By the way, process traces could also have been kept, so at this point the programmer may also wish to consult these traces.

After examining the state and traces, the programmer can set other breakpoints and resume operation of the system. The latter is achieved by broadcasting "continue" messages to all computers.

As an alternative, the programmer may wish to execute one instruction only and then go back to examining the state and traces. First off, the programmer must decide which one of the processes should be the one to execute its next instruction. For this decision, the master debugger must be able to display a list of all the active and ready processes on the master terminal. (An active process is one that was running when system execution was suspended. A ready process is one which is ready to run but was not actually running. Waiting processes are suspended waiting for some condition to exist. Only ready and active processes can execute their next instruction.) When the programmer selects a process, the master debugger instructs the computer which runs it to execute its next instruction only. After that, the programmer is back in examination mode.



NODE STATUS

NODE:	1	2	3	4
LAST TIME:	10:00	10:05	9:57	7:01
CPU UTILIZ:	97	52	10	-
AVE QUEUE:	17	21	3	-
#PROC:	5	7	2	0
STATUS:	OK	OK	OK	DEAD

PROCESS STATUS

PROC:	A/1	B/1	X/3
STATUS:	OK	SUSP.	OK
%CPU:	10	0	15
MESSAGES PER SEC:	2	0	.7
TOTAL MESSAGES:	151	37	841

TRANSACTION DISPLAY

TRANS. ID.	ORIGIN NODE	ACTIVE NODE	STATUS	MESSAGES
437.1	1	4	RUNNING	5
41.3	3	3	INIT	0
111.2	2	1	COMMIT	10

SPECIAL MESSAGES:

FROM: PROC. Z/2; TIME: 9:52; NODE:2;
EVENT: DEADLOCK FOUND
PARTICIPANTS: B/1 - C/1 - T/2 - B/1; VICTIM: B/1

COMMAND?

Fig. 3. A monitoring display.

In many cases, executing a single instruction may not be very productive. A better approach would be to run a process until its next "significant" event. (These are the same events we are tracing.) From the point of view of the interaction among the processes, nothing much occurs in a process between significant events. Therefore, to find bugs which result from the interaction of processes, it makes sense to run a process until such an event occurs. Thus, when a programmer selects a process to be run, he or she can be given the option of executing a single instruction or of running the process until its next significant event (or until the next significant event of a certain type).

D. Monitoring

After testing and debugging a distributed system for some time, we may reach a stage where the system seems to be running. At that point, we probably cannot afford to examine every trace that is produced by the system. Still, we would like to "keep an eye" on the system to make sure that things are running smoothly. The monitoring facility which we will describe in this section can be useful for this purpose.

The main idea is to provide the programmer on his or her master terminal with a real-time summary of the system execution. This overall picture should give the programmer information such as what transactions are being executed, where the transactions are being executed, the status of the processes and nodes, the message traffic, and so on. This information will be helpful in observing the system performance and in discovering bottlenecks or other abnormal behavior. Fig. 3 illustrates one such type of display.

What type of information should be displayed on the master terminal? One type is performance information. This information can be collected by the operating system module at each computer, and includes values such as average queue sizes, CPU utilization, and process status. Another type of useful information is contained in the "important" events that are registered in the process traces. From this information we could obtain the status of the transactions, the number of deadlocks detected, the rate of file accesses, and so on (see Fig. 2).

Thus, one way in which the system monitoring could take place is as follows. At his or her master terminal, the programmer selects the information which is of interest. That is, the programmer builds a list of (predefined) performance measures and a list of important event types. These lists are distributed to all computers. The operating system modules then start collecting the requested performance information and periodically send it to the master debugger. The list of event types which was provided by the programmer is used to filter the important events, and only data on the selected ones is forwarded to the master debugger.

At the master debugger, all the information arriving from the computers must be displayed in some meaningful fashion. A general purpose display package could be utilized for this, but in many cases it would be best to let the programmer display the collected information as he or she pleases. If this approach is used, the programmer would write a display program which would receive the monitoring information in a standard predefined format.

When undesired behavior is observed in monitoring mode, the identification of the bug is clearly more difficult. Since there are no traces, the system will have to be tested, with tracing or controlled execution, until the error is reproduced.

V. Conclusions

In this paper we have discussed the issues involved in debugging a distributed computing system. We argued that distributed debugging is significantly different from sequential program debugging and hence merits special attention. We looked at debugging techniques which would be appropriate for distributed systems (i.e., bottom-up debugging, tracing program execution, and two-phase debugging). We also suggested debugging tools for producing and examining traces, for controlled system execution, and for monitoring.

The ideas presented in this paper were developed while the authors implemented, tested, and debugged an experimental distributed transaction processing system [11]. (The system was implemented in the Corporate Research Group of Digital Equipment Corporation.) Each computer in the system contains one or more databases, and the users of the system are allowed to select and run transactions from a set of predefined transactions. A single transaction can access and modify data located at different computers. Such a system can be used for an airline reservation system, or say, a banking system.

For this project, we did not develop any general purpose debugging tools. Instead, some of the debugging aids we discussed in this paper were built into the system as it was written. Each process has a trace file associated with it. The low level modules in the system (e.g., the message handling module) automatically record important events on this file. In addition, the application process also writes significant state changes in the file. We did not write any querying programs to examine the trace files. We only used a standard screen text editor (with limited string matching capabilities).

Even in this limited fashion, we found the trace files to be extremely helpful. Searching through the files was slow (especially because the files were on different machines), but the summarized data contained in the files really captured the essence of what had occurred in the test run. In many cases, we pinpointed the exact cause of a bug in the first debugging phase.

In our system we did not implement any facilities for controlled system execution. However, at a few points in the development of the system such facilities would have been very useful. (Only a lot of patience and time took the place of these facilities.)

We have implemented a monitoring facility similar to the one described in this paper. This facility is also being used to collect statistics on the system performance (e.g., the average execution time of transactions of a certain type). Thus, the monitoring facility is not only being used to identify abnormal behavior, but also for demonstrating the system and for performing experiments. (Some preliminary results comparing several database locking policies can be found in [15].)

Distributed computing systems have come into widespread use only recently. Experience with implementing and debugging them is imited, so we can expect better debugging strategies and tools in the future. In the meantime, the ideas presented here may serve as a guide for the programmer faced with the task of debugging a distributed system.

Acknowledgment

Many useful ideas and suggestions have been provided by S. Davidson, J. Kent, R. Lipton, I. Nassi, K. Pammett, L. Randerson, E. Van Horn, M. Chung, K. Wilner, and the referees.

References

[1] R. M. Balzer, "EXDAMS–Extendable debugging and monitoring system," in *Proc. Spring Joint Comput. Conf.*, Montvale, NJ, 1969, pp. 567–580.

[2] P. Brinch Hansen, *Operating System Principles.* Englewood Cliffs, NJ: Prentice-Hall, 1973.

[3] —, "Testing a multiprogramming system," *Software–Practice and Experience,* vol. 3, pp. 145–150, 1973.

[4] A. R. Brown and W. A. Sampson, *Program Debugging.* New York: American Elsevier and MacDonald, 1973.

[5] E. F. Codd, "A relational model of data for large shared data banks," *Commun. Ass. Comput. Mach.*, vol. 13, pp. 377–387, June 1970.

[6] J. Cohen and N. Carpenter, "A language for inquiring about the run-time behavior of programs," *Software–Practice and Experience,* vol. 7, pp. 445–460, July–Aug. 1977.

[7] O. J. Dahl, E. W. Dijkstra, and C. A. Hoare, *Structured Programming.* New York: Academic, 1972.

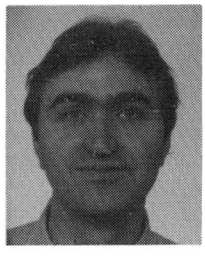
[8] C. J. Date, *An Introduction to Database Systems,* 3rd ed. Reading, MA: Addison-Wesley, 1981.

[9] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages,* F. Genuys, Ed. New York: Academic, 1968.

[10] P. H. Enslow, "What is a distributed data processing system?" *IEEE Computer,* pp. 13–21, Jan. 1978.

[11] H. Garcia-Molina, F. Germano, and W. Kohler, "Architectural

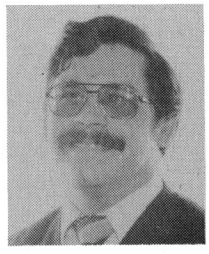
overview of a distributed software testbed," in *Proc. 16th Hawaii Int. Conf. Syst. Sci.*, vol. I, Jan. 1983, pp. 310–319.
[12] J. N. Gray, "Notes on database operating systems," Advanced Course on Oper. Syst. Principles, Tech. Univ. Munich, July 1977; also in *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmuller, Eds. Springer-Verlag, 1979, pp. 393–481.
[13] R. M. Keller, "Formal verification of parallel programs," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 371–384, July 1976.
[14] D. E. Knuth, *The Art of Computer Programming*, vol. 1. Reading, MA: Addison-Wesley, 1973, p. 189.
[15] W. Kohler, K. Wilner, and J. Stankovic, "An experimental evaluation of locking policies in a centralized database system," in *Proc. SIGMOD Conf.* San Jose, CA, May 1983.
[16] L. Lamport, "Time clocks, and the ordering of events in a distributed system," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 558–564, July 1978.
[17] S. Lauesen, "Debugging techniques," *Software–Practice and Experience*, vol. 9, pp. 51–63, Jan. 1979.
[18] Z. Manna, *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.
[19] G. J. Myers, *Software Reliability–Principles and Practices*. New York: Wiley, 1976.
[20] S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 279–285, May 1976.
[21] J. B. Rothnie and N. Goodman, "A survey of research and development in distributed database management," in *Proc. 3rd VLDB Conf.*, Tokyo, Japan, 1977, pp. 48–62.
[22] J. T. Schwartz, "An overview of bugs," in *Proc. Courant Comput. Sci. Symp.*, vol. 1 (*Debugging Techniques in Large Systems*). Englewood Cliffs, NJ: Prentice-Hall, 1970, pp. 1–16.
[23] R. M. Spunik, "Debugging under simulation," *in Proc. Courant Comput. Sci. Symp.*, vol. 1 (*Debugging Techniques in Large Systems*). Englewood Cliffs, NJ: Prentice-Hall, 1970, pp. 117–136.
[24] J. A. Stankovic, "Debugging commands for a distributed processing system," in *Proc. COMPCON Fall 1980*, pp. 701–705.
[25] J. D. Ullman, *Principles of Database Systems*. Comput. Sci. Press, 1980.
[26] E. C. Van Horn, "Three criteria for designing computing systems to facilitate debugging," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 360–365, May 1968.
[27] D. Van Tassel, *Program Style, Design, Efficiency, Debugging and Testing*. Englewood Cliffs, NJ: Prentice-Hall, 1974.



Hector Garcia-Molina received the B.S. degree in electrical engineering from the Instituto Technologico de Monterrey, Monterrey, Mexico, in 1974, and the M.S. degree in electrical engineering and the Ph.D. degree in computer science, both from Stanford University, Stanford, CA, in 1975 and 1979, respectively.

He is currently an Assistant Professor in the Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ. His research interests include distributed computing systems and database systems.

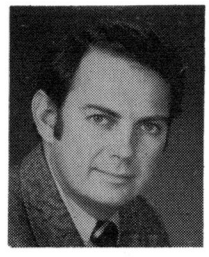
Dr. Garcia-Molina is a member of the Association for Computing Machinery.



Frank Germano, Jr. (M'79) received the B.S. degree in engineering from Cornell University, Ithaca, NY, in 1969, the M.B.A. degree in business administration from Stanford University, Stanford, CA, in 1971, and the Ph.D. degree in computer and information sciences from the University of Pennsylvania, Philadelphia, in 1980.

During 1972 he was manager of user services for the research computer facility at Stanford Medical Center. From 1973 to 1974 he was manager of systems and programming for GTE Information Systems in Mt. Laurel, NJ. During 1975 he developed MUMPS-based, medical database systems for the Department of Radiation Therapy at Thomas Jefferson University Hospital, Philadelphia, PA. From 1976 to 1979 he was on the Computer Science faculty at Temple University and a database consultant to the American College of Radiology, Philadelphia. From 1979 to 1981 he was manager of the distributed software research group within the corporate research group of Digital Equipment Corporation. He is currently an engineering manager at Apollo Computer, Inc., Chelmsford, MA. His technical interests include distributed data management, network operating systems, and graphics-based user interfaces in the workstation network environment.

Dr. Germano is a member of the Association for Computing Machinery, the IEEE Computer Society, and the American Association for the Advancement of Science.



Walter H. Kohler (S'66–M'73) was born in Dover, NJ, on June 2, 1945. He received the B.S.E., M.S., M.A., and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ, in 1967, 1968, 1970, and 1972, respectively.

From 1968 to 1969 he was a member of the technical staff of Bell Telephone Laboratories, Holmdel, NJ. Since 1972 he has been with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, where he is currently an Associate Professor. During his tenure at the University of Massachusetts he has held a wide variety of academic and technical administrative positions, most recently serving as Acting Department Head for the 1982/1983 academic year. Since 1979 he has also been associated with the Corporate Research and Architecture Group, Digital Equipment Corporation, as a consultant. His current research interests are in the area of distributed transaction and database system design and implementation, with particular interest in algorithms for concurrency control and recovery and performance measurement, modeling, and evaluation. His previous work has included analytical and numerical studies of exact and approximate algorithms for scheduling and resource allocation problems.

Dr. Kohler is a member of the Association for Computing Machinery.