ONLINE DETECTION OF DISTRIBUTED PREDICATES IN DISTRIBUTED PROGRAMS

APPROVED:

Supervisor: ______________________________

VIJAY KUMAR GARG

____

MOHAMED GOUDA

ONLINE DETECTION OF DISTRIBUTED PREDICATES IN DISTRIBUTED PROGRAMS

by

GREG MICHAEL HOAGLAND, B.S.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

August, 1992

Acknowledgments

I want to thank the following people: Dr. Garg, my friends in ENS 512 and Dr. Gouda for taking the time to read my thesis.

GREG MICHAEL HOAGLAND

The University of Texas at Austin

August, 1992

ABSTRACT

ONLINE DETECTION OF DISTRIBUTED PREDICATES IN DISTRIBUTED PROGRAMS

by

GREG MICHAEL HOAGLAND, B.S.

SUPERVISING PROFESSOR: VIJAY KUMAR GARG

This thesis presents an online debugger for detecting distributed predicates in a distributed program and an environment for creating event-driven distributed programs that can be debugged by our online debugger. Detecting predicates in a distributed program is a difficult problem because the global state is divided among the processors in the distributed system. Also, some of the predicates that are well-defined for a sequential program are ill-defined for a distributed program. We use a special logic for defining the syntax and semantics of distributed predicates. Our debugger implements a set of distributed algorithms

that is able to detect online a subset of these predicates in a distributed program. Once a predicate is detected the distributed computation can be halted in a global state where the predicate is true. The debugger has been developed using C on a Sun3/80 under SunOs4.0.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis presents the design and implementation of an online debugger used to debug distributed programs by detecting distributed predicates.

1.1 Distributed Systems and Motivations For Using Them

A distributed system consists a set of processors that do not share memory or share a clock. Processors communicate by sending messages over an interconnection network. These type of systems are sometimes referred to as loosely coupled systems because communication between processors is explicit[19]. This is in contrast to shared memory parallel systems, sometimes called a tightly coupled systems. Since the processors share memory, they are able to communicate by simply accessing memory. Concurrent programs can be written for both types of systems, but we will only be looking at debugging those written for distributed systems because despite their similarities, there exist many fundamental differences that make the problems of debugging quite different.

With the increasing popularity of the local-area networks (LAN), distributed systems are becoming widespread. Some of the motivating reasons for using distributed systems are the following: resource sharing, fault-tolerance, computation speedup and communication. It is likely that distributed systems will become even more important with the emergence of wide-area networking which is an offspring of the merger of computer networking with telecommunication networking. Predictions by many people are that the distributed applications running on wide-area networks will become a very integral part of people's everyday lives much like the telephone networks are today[5].

1.2 Distributed Program versus Sequential Program

The reason it is more difficult to create a distributed program stems from the following factors: partial ordering of process states, multiple threads of control and distributed global state.

A fundamental property of a distributed program is that events are not totally ordered. Instead, the events exhibit a partial ordering[11]. This lack of a unique total ordering of events can make it very difficult to build and debug a distributed program. For example, two executions of a program with the same partial orderings can exhibit different behaviors as a result of a different ordering of events, and a condition that might indicate an error in one ordering may not even be observed in the other.

Because there exists concurrent threads of control in a distributed application, it is often necessary to synchronize the threads. An example is a distributed file system with replicated data that enforces a one writer-multiple reader policy. The nondeterminism of a distributed program makes it difficult to properly test a synchronization algorithm. This is evidenced by the fact that it is not uncommon to see counterexamples published describing the deficiencies of a synchronization algorithm[9].

Additionally, the fact that the global state is divided among processors in the network makes building distributed applications difficult. An example is a distributed mutual exclusion program. When a process is in its critical section, it has no way of knowing what the current local state of the other processes are. Therefore, the programmer must use tricky synchronization algorithms to restrict the distributed program from entering a subset of the global state space where mutual exclusion is violated.

1.3 Debugger for Distributed Programs

It is unrealistic to think that the need for debugging systems will be eliminated any time soon. Software systems are becoming so complex it is only natural to expect that errors can and will exist in the system. If that were not the case, testing would not be such an important step in the software development process.

Two ways debugging can be done is *online* or *offline*. Offline debugging, or post-mortem debugging as it is sometimes referred to, begins after the distributed computation has completed execution. As the program executes, the processes are monitored so that interesting events can be logged to trace files. When the program finishes, the offline debugger examines the traces files for interesting events. An online debugger, on the other hand, tries to detect the events on-the-fly as the program is executing.

Unfortunately, any program that is being debugged dynamically, whether it is done online or offline, is influenced by the *probe effect*. What this means is that watching or monitoring a program will perturb its behavior. For sequential systems, the probe effect is usually of no concern. This is not true for concurrent programs where the probe effect can have very serious consequences. The amount of perturbation depends on how intrusive the monitor is. Minimizing intrusion requires special customized monitoring hardware that is able to passively watch a program. Software monitors, such as the one we use, run on the same processors as the program and are usually very intrusive.

For this thesis, we have built an online debugger for distributed programs. In the process, we have developed a programming environment for developing event-driven distributed programs. We utilize a logic presented by Waldecker and Garg in [25] that defines distributed predicates whose truthness depends upon the global state or sequence of global states from a run of a distributed program. The predicates are unstable in the sense that consistent global snapshots[2] are not guaranteed to detect the occurrence of the predicates.

The debugger can be used in ways that are directly analogous to how a single process sequential program would be debugged. For instance, suppose you are debugging a sequential program where you know that variable x should never be greater than 10 while variable y is less than 5. The GNU gdb debugger provides the ability to set a breakpoint when

$$(x > 10) \wedge (y < 5).$$

Similarly, suppose we have a distributed program and we know that it should never be true that process 1 and process 2 are in their critical sections(CS) at the same time. We can set the following breakpoint:

$$(\text{process 1 is in } CS) \wedge (\text{process 2 is in } CS).$$

If this breakpoint is ever detected, it indicates that there exists an error in the program. However, while the conjunctive operator is well-defined in a sequential system where all events are totally ordered and the address space of the process is centralized, that is not the case for a distributed program where the events are partially ordered and the address space is distributed. Waldecker's logic presents a conjunctive operator and its semantic meaning for a run of a distributed program.

There can exist a substantial delay between the occurrence of a distributed predicate and its detection[20]. In fact, by the time the predicate is detected and the distributed program is halted, the program may no longer be in a state that is interesting to examine. Therefore, for one type of distributed predicate, we have provided a rollback and halting mechanism that will stop the distributed program in a global state where the distributed predicate is satisfied.

1.4 Outline of This Report

This report is broken down as follows. Chapter 2 will present related work in debugger development for concurrent programs. Chapter 3 will present the model of a distributed program as well as the logic used to define distributed predicates whose satisfaction depends upon the global state or sequence of global states of a run. It will also present the types of distributed predicates that our debugger will be able to detect along with the detection algorithms.

Chapter 4 will present the design of the environment we use to develop event-driven distributed programs. Also, it will present the design of the debugger used to debug the event-driven distributed programs online. Chapter 5 will present some examples of distributed programs that we have developed along with some examples of how the debugger can be used to debug these distributed programs. Discussions, conclusions and suggestions for future work are given in chapter 6. The Appendix includes the source code for the debugger. It is split into three sections that correspond to the three main modules of the debug environment. The appendix also includes a manual that describes how to use the online debugger.

Chapter 2

Background and Related Work

In the case of debuggers for sequential or single process programs, the list of features that they are expected to include is relatively well-defined. They should allow the programmer to specify breakpoints that halt execution on a line, function or when a predicate has been detected. Additionally, they should allow a program to be traced or single-stepped and allow memory to be examined and modified.

Sequential debuggers are used to monitor a program's execution for a behavior or state which can be specified with a predicate defined on the state space. Also, and maybe more importantly[12], they are used to examine a program's behavior over a sequence of states. The former is achieved with predicate breakpoints, whereas the latter is achieved using a rather primitive method of single-stepping in the interesting portions of the program and examining the memory after each step to get an idea of the program's temporal behavior.

We believe that a similar process would be useful to debug distributed programs. However, the underlying properties of distributed programs discussed in chapter 1, such as lack of a unique total ordering of events and distributed global state, make it much more difficult. It has thus been necessary to rethink how debuggers should be built and designed to compensate for the additional complications. In this section, we will present the avenues that researchers are exploring to facilitate debugging of concurrent programs. This is a very active research area and a wide range of approaches are being taken to tackle the problem[16]. Some of them are discussed below but for a more comprehensive list and survey of many of these efforts see [14].

2.1 Characterize Correct Behavior

One approach to debugging is to specify the correct behavior of the system and then let the debugger analyze the system, either statically or dynamically, to see if the correct behavior is ever violated.

A static approach is presented by Emerson in [6]. A concurrent program is represented as a finite state automata. Specifications for the program are expressed using a temporal logic language called CTL. The system works by unrolling the finite state automata into a tree where each branch represent a possible execution path the computation may proceed along. Next, the system verifies whether or not the specifications hold for the given tree.

The operators in a temporal logic, such as CTL, provide a very intuitive and powerful way of expressing specifications and conditions in a program, for example, "p until q in all paths". Therefore, they have been incorporated into the logic developed by Waldecker and Garg which is used to define the distributed predicates detected by our debugger[25]. The logic is presented in chapter 3.

While the system in [6] statically analyzes a concurrent program, in [1] Bates proposes a method of describing the desired behavior of a program in terms of events which are checked against the actual behavior during run-time. Events, described using a language called *Event Description Language* (EDL), are built from primitive interprocess events, such as *send message* and *receive message*, and from previously defined events. By describing events recursively, it becomes possible to view the behavior of the system in terms of high-level abstract events, a process which Bates terms *behavioral abstraction*.

Behavioral abstraction is a very nice feature of the system because it frees a programmer from having to manually infer the high-level behavior of a distributed computation from the the uninteresting barrage of individual low-level primitive events. The EDL language uses the operators of concatenation which signifies sequential operation, choice, kleene closure, positive closure and an interleaving shuffle when defining events in term of existing high-level or primitive events.

Bates proposes behavioral abstraction or "event-based" debugging as an alternative to "state-based" debugging. The program's observable behavior is viewed in terms of interprocess interactions instead of process states or global states. Below is a simple example showing how a high-level event called *secure_login* may be built from the lower-level events *encrypt_message* and *login*. The event represents the procedure for securely logging in from one process to another. The events are written as regular expressions using the concatenation operator, $*$, which means the operand events are executed sequentially in the order they are written. This is an interesting example because it illustrates how a programmer can describe a relatively abstract event such as a *secure login* and how the debugger recognizes the high-level event from lower-level events. In essence, the debugger becomes an automatic filtering mechanism for the programmer.

$$\begin{aligned}
encrypt_message(P_1, P_2) &= P_1 \text{ sends message to encryption server} * \\
&\quad \text{message encrypted with } P_2's \text{ key} * \\
&\quad \text{return encrypted message to } P_1; \\
login(P_1, P_2, enc) &= P_1 \text{ sends a message id and password to} P_2 \\
&\quad \text{in encrypted or unencrypted mode;} \\
secure_login(P_1, P_2) &= encrypt_message(P_1, P_2) * \\
&\quad login(P_1, P_2, encrypt);
\end{aligned}$$

Hseush and Kaiser's approach of *data path expression* is another way of comparing actual program behavior versus expected program behavior[10]. All the systems proposed in this section suffer from the same problem. The programmer must be able to foresee an exhaustive list of possible errors and, from this list, develop a set of events that enables the debugger to detect them.

2.2 Record and Replay Events

A big problem when testing or debugging a distributed system is the nondeterminism in execution. The observed behavior in a sequential program with a fixed input is repeatable from one execution to the next which is not the case for distributed programs. Even if one has an exhaustive set of inputs, one may not detect all errors because of the nondeterminism inherent in a distributed computation.

One method of countering this reproducibility problem is to record or log the interesting events in the computation, such as message *sends* and *receives*, to create an *event history* for each process. The debugger then uses these event histories to control the reexecution of the distributed computation. The debugger can reexecute the processes using different relative execution speeds, and when erroneous behavior is observed in an execution, the program can be replayed and examined in detail to find errors.

One method of maintaining these event histories is to record the entire content of received messages to a log. Then, using a sequential-like debugger, a single process can be replayed under the control of the event history. As long as the event history correctly reflects the interprocess communication, the replay will produce the same results as the original execution. This reduces the problem to that of debugging a sequential process once a faulty process has been observed in the distributed computation[14].

One drawback to maintaining event histories in the method described above is that the space overhead required to maintain the entire content of the messages may become too costly especially when a lot of uninteresting events are being recorded. Some systems counter this problem using filters that ignore uninteresting events. The *Recap* debugger by Pan reduces event histories by recording and logging messages only during the interesting segments of the execution. A checkpoint is taken just before the interesting segment so the debugger can start replaying from there[17].

The *Instant Replay* debugger by Leblanc and Mellor-Crummey is another attempt at reducing the size of event histories. Their debugger stores only the order that messages are received without message contents in the event history. Upon replay, the debugger enforces a specific ordering of messages in the reexecution as dictated by the event history. This enables the distributed computation to recompute the original messages' contents. One implication of not storing the message content in the event history, however, is that a single process cannot be debugged in isolation as is possible in Recap.

2.3 Visualization and Auralization

An important goal of research in concurrent debugging systems is to find ways of presenting distributed data and control of concurrent programs to users in a manner that helps improve understanding. Visualization and auralization are two areas where research into improving presentation is being examined. For example, Jackson suggests that the human mind has a tremendous capacity for processing sound information[16]. In some proposed debuggers and program monitors, aural data is presented in a passive manner that allows anomalies to be detected by the programmer without requiring them to pay strict attention to the sounds. It is based on the same principle that a person's attention is automatically alerted when he hears his name called.

The four most common techniques for presenting data visually are the following: *textual*, *process-time diagrams*, *animation* and *multiple windows*. Textual refers to what and how textual data is presented by the debugger, for example, the effects of using different colors or how textual placement effects data comprehension.

Process-time diagrams illustrate how the concurrent computation behaves over time. Figure 2.1 is an example of a typical process-time diagram.

Research in the area of animation is examining ways to represent the current state of a computation visually on the screen. Since a frame of the animation represents only one state of the concurrent computation, it is possible to present more information per state than is possible with a process-

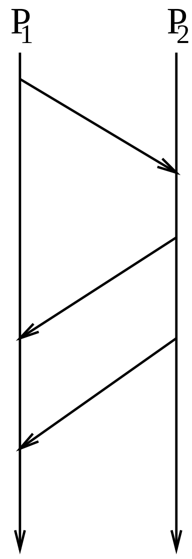

Figure 2.1: A process-time diagram

time diagram. Animation is good at detecting a global state whereas process-time diagrams are better at presenting the temporal behavior of a program.

This leads us into the last technique for visually presenting data which is to use multiple windows. This has become a natural way of presenting data from concurrent computations since they consist of multiple processes and because graphical user interfaces, such as X-windows, have made it easy to display multiple windows on one viewing monitor. A debugger, for example, might report status information from each process in a separate window. This is in fact similar to how data is presented in our debugger. We have a separate monitoring window for each process and another one for the debugger's coordinator.

2.4 State-based Debugging

This is the type of debugger we have designed and implemented. Of all the debuggers discussed so far, the state-based debuggers are used in a way that is most analogous to how today's sequential debuggers are being used. They are used to examine and monitor the state of a distributed computation. Some of the issues of concern are detecting global states or sequences of global states and recreating or rolling back global states in distributed computations. The remaining part of this chapter will cover interesting and related work that is being done in the area of state-based debugging for distributed programs.

2.4.1 Miller and Choi

In [15], Miller and Choi have defined *distributed breakpoints* as the detection of a predicate followed by the halting of the distributed program in a consistent global state, which is a global state such that none of the local states are ordered. They define four types of predicates. They are:

- *Simple Predicates:* These are predicates, SP, that are based on the state of a single process
- *Disjunctive Predicates:* These are predicates composed of simple predicates joined by the disjunctive operator, $\cup$. $DP = SP[\cup SP]*$
- *Linked Predicates:* These are predicates composed of disjunctive predicates joined by Lamport's happened before relation. $LP = DP[< DP]*$
- *Conjunctive Predicates:* These are predicates composed of simple predicates joined by the conjunctive operator, $\cap$. $CP = SP[\cap SP]*$

The first three predicates listed above are well-defined, and Miller and Choi have presented algorithms for detecting them. For a sequential program, a conjunctive predicate is satisfied the instant all the simple predicates are satisfied, but since the notion of a global clock does not exist in a distributed system, it may not be possible to know whether or not all the simple predicates are satisfied at the same instant. As a result, Miller and Choi have provided their own semantics for a conjunctive predicate. It is satisfied after all the simple predicates that form the conjunctive predicate have been satisfied regardless of whether or not the local states where the simple predicate were true are ordered according to Lamport's happened before operator[11].

Although this is a useful interpretation in some circumstances, it is inappropriate for others. For example, suppose we would like to check if two processes are in their critical section at the same time. One would like to use a conjunctive predicate with an interpretation that is close to the traditional interpretation used in a sequential system, such as, is it possible that the two processes are in their critical sections simultaneously? In the case of Miller and Choi's interpretation of conjunctive predicate,

$$CP = (P_1 \text{ in CS}) \cap (P_2 \text{ in CS})$$

would be detected even if P_1 and P_2 were in their critical sections at different times under a correct mutual exclusion algorithm.

As described above, the complete definition of the distributed breakpoint calls for stopping the distributed computation in a consistent global state once the breakpoint predicate is detected. For stopping the program, Miller and Choi have presented a halting algorithm that is a variant of Chandy and Lamport's global snapshot algorithm[2].

2.4.2 Haban and Weigel

In [8], Haban and Weigel define a *global breakpoint* as the following sequence of actions: detection of a *global event* associated with the breakpoint followed by the halting of the distributed computation in a consistent global state.

Global events are composed of one or more *primitive events* or previously defined global events thus making the global event definition recursive. A primitive event is satisfied by a single process. The relations between the events are the following:

- *Alternative:* $Ae = Ge_1 | Ge_2$. The alternative global event, Ae is satisfied when either Ge_1 or Ge_2 is satisfied. Ae inherits the vector time of first event that occurs.

- *Conjunctive:* $Ce = Ge_1 \& Ge_2$. The conjunctive event, Ce is satisfied after both of the global events Ge_1 and Ge_2 are satisfied. Ce inherits the vector time of the last event that occurs.

- *Happened-Before:* $He = Ge_1 < Ge_2$. The conjunctive event, He, is satisfied if Ge_1 is causally ordered before Ge_2. He inherits the vector time of Ge_2.

- *Simultaneous:* $Se = Ge_1 \hat{} Ge_2$. The simultaneous event, Se, is satisfied if event Ge_1 is causally unrelated to Ge_2. Se inherits the vector time of the last event to occur. This rule for assigning timestamps is not intuitive and can make interpreting simultaneous global events confusing.

The following is an example showing why their method of assigning timestamps in the case of simultaneous events is confusing. Suppose we are trying to detect the simultaneous event

$$Se = (SP_1 \ \hat{}\ SP_2) \ \hat{}\ SP_3$$

The expression tree for evaluating Se is shown in Figure 2.2. Intuitively, one would think that Se would be satisfied only if the events SP_1, SP_2 and SP_3 can occur simultaneously. However Figure 2.3 shows that this is not necessarily true. In space-time diagram a), assume for the sake of simplicity that the vertical time axis represents real-time. Since events SP_1 and SP_2 are causally unrelated, node 3 in the expression tree is satisfied and is assigned the vector time from SP_1 according to the timestamping rules for simultaneous events. Next, when SP_3 occurs, its vector time is compared against that of node 3. Since the vector times are simultaneous because SP_1 and SP_3 are causally unrelated, Se is satisfied. This is true despite the fact that it is not possible for the three events to be concurrent since SP_2 and SP_3 are causally related. In diagram b) Se is not satisfied because node 3 is assigned the vector time of SP_2 instead of SP_1.

Like Miller and Choi, Haban and Weigel halt the program using a variant of Chandy and Lamport's distributed snapshot algorithm.

They have implemented the mechanisms for specifying and detecting global events and halting a system in a consistent state in a debugger called

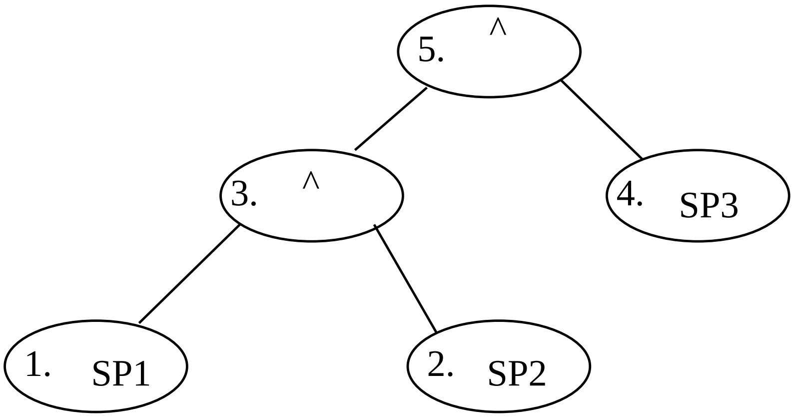

Figure 2.2: Expression tree for evaluating Se.

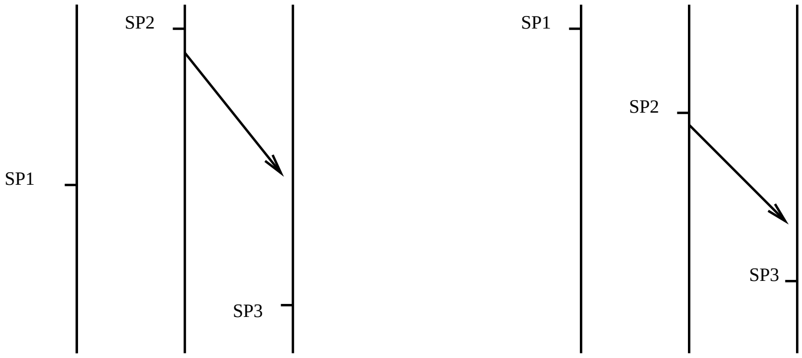

Figure 2.3: Space-time diagrams: a) Se satisfied b) Se not satisfied

Distributed Test Methodology (DTM). A special hardware test architecture has been developed to monitor the processes in the distributed system which helps to minimize the perturbation effects when debugging.

2.4.3 Spezialetti

In [21], Spezialetti and Kearns argue with the notion of using logical clocks to get a concurrent view of operation in a distributed systems. They claim that the lack of causality between events provides no type of unifying relationship between the events and thus it is inappropriate to use as the basis for saying two events are simultaneous. So instead, they present a iterative partial snapshot algorithm to split the distributed computations into simultaneous regions. For each simultaneous region, they determine the state of a distributed event by applying the event's operator, such as conjunction, to the event's operands.

The strengths of using simultaneous regions are that they are able to detect global predicates of the form:

$$(x_1 + x_2 + x_3) > CONSTANT$$

where the variable x_i can reside on different processors. They detect it by checking the variables in each simultaneous region to see if the sum is greater than $CONSTANT$. However, a major weakness of their work, as pointed out by Cooper in [4], is that their algorithms are incomplete. One can present examples to show that if their algorithm does not detect an event then this is not sufficient to say that the event does not occur.

In [20], Spezialetti discusses the problem of minimizing the latency between the occurrence of an event and its detection. She examines how the properties of the event's operands and operators as well as the semantics of the computation can be used to select the most effective monitoring strategy, such as deciding where to place the event evaluation monitors which are the modules that collect an event's component information, organize the information into a consistent view and then evaluate if the event was ever true.

2.4.4 Fowler and Zwaenpoel

In [7], Fowler and Zwaenpoel look at the problem of a halting a distributed computation after an event is detected. Ideally, one would like a distributed computation to halt immediately when the event occurs. However, halting algorithms based on Chandy and Lamport's distributed snapshot algorithm cannot immediately halt a distributed program, but even if they could, it still might not be possible to halt the program in a state where the event occurs since the delay between event occurrence and detection can be substantial. As a result, when a program is finally halted, it may be in a state that gives no indication why a breakpoint event occurred.

Fowler and Zwaenpoel define a *causal distributed breakpoint*. This is a breakpoint that is initiated by a "sequential breakpoint event" within one of the processes. When the breakpoint event occurs, all other processes in the system are then restored to their earliest state that reflects all events that happened before the breakpoint event.

Figure 2.4 presents an example of a causal distributed breakpoint.

The breakpoint event is represented by the X in P_3. Once detected, the debugger calculates the state the distributed program should be halted in, which in this case is at 1 in P_1 and 2 in P_2. Getting the system in this state may require that the processes be rollbacked, so Fowler and Zwaenpoel have incorporated a checkpoint and message logging capability into their debugger.

By stopping in a global state that reflects all events that happened before the breakpoint event, each process is in a local state that is just beyond the last state that could have had a causal effect on the breakpoint event. These local states may indicate to the programmer why the breakpoint occurred, whereas if the processes are halted in a later state, they may move out of a state that indicates the cause of the breakpoint event.

We have utilized a variant of the causal distributed breakpoint in the online debugger we have built. Instead of requiring the breakpoint event to be local to a single process like Fowler and Zwaenpoel, our breakpoint event can be a distributed predicate.

2.4.5 Cooper and Marzullo

In [4], Cooper and Marzullo provide a model for a distributed computation. The possible global states form a lattice-like structure. Cooper and Marzullo also define the temporal operators, *possibly* and *definitely.* The temporal operators are used in conjunction with a global property, Φ. For example, *Possibly* Φ is satisfied if there exists a possible execution path in the distributed computation such that global property Φ is true, and *Definitely* Φ is satisfied if Φ is true in all possible execution paths.

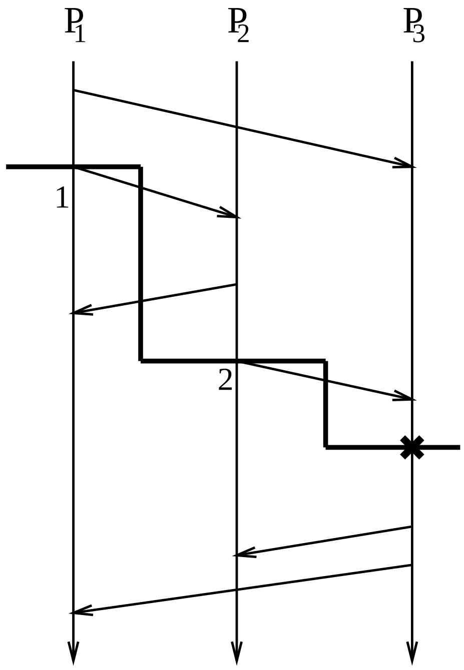

Figure 2.4: Breakpoint event occurs at X in P_3. The causal breakpoint algorithm would halt P_1 at 1 and P_2 at 2.

Definitely and possibly are very closely related to the way in which we are using the terms *strong* and *weak*[24]. For example, we define a strong conjunctive predicate as one that is always true for a given execution, and weak conjunctive predicate as one that can be true for a given execution.

The algorithmic approach taken by Cooper and Marzullo to detect distributed predicates is similar to Chin's approach in [3] of searching the lattice space of global states. Both approaches are able to detect a wide variety of distributed predicates. However, Cooper and Marzullo's model of computation is less general. When modeling the transition between global states, only one local process is allowed to advance its state which forces the global computation to proceed in a sequential lockstep fashion and does not allow for the notion of concurrency of local process events in a global step.

Chapter 3

Detecting Distributed Predicates

This chapter presents the assumed model for a distributed program. Additionally, we will discuss Waldecker's logic for defining distributed predicates, and finally the chapter covers the algorithms used to detect the distributed predicates that our online debugger detects.

3.1 Model of Distributed Program

A distributed program consists of a collection of processes. The processes communicate using asynchronous messages which means that sends are non-blocking and receives are blocking. Message channels can be non-FIFO and non-reliable. An execution of a distributed program is modeled using a state-transition based model.

Definition 3.1 *A* local state *is a state of a single process. A* global state *of a distributed program is a state that is composed of a local state of each process.*

Each process is modeled by a *process trace*, $s_0 s_1 \cdots s_n$. This is a sequence of local states where s_i represents the *ith* local state that is entered by the process. Transitions between local states are caused by *events* in the process. These events can be classified as *internal*, *send*, or *receive*. An example of an internal event is a variable assignment. Send and receive events correspond to a process sending and receiving a message respectively.

The events that cause a transition between local states can be ordered by Lamport's "happened before" operator[11]. For notational convenience, we present the ordering relation in terms of the local states that are the result of an event.

Definition 3.2 *For two local states, s_a and s_b, we say $s_a < s_b$ if and only if:*

1. *s_a and s_b are in the same process trace and the index of s_a is smaller than the index of s_b, OR*
2. *A message m is sent in s_a and received in s_b, OR*
3. *$\exists\, s'$ such that $s_a < s' \wedge s' < s_b$*

Definition 3.3 *Given local states s_i and s_j from the process traces, s_i and s_j are* comparable *if and only if*

$$(s_i < s_j) \vee (s_j < s_i).$$

Otherwise the states are incomparable

The set of local states in the process traces are partially ordered.

The concurrent execution of the processes is modeled by a *global trace* which is a sequence of global states, $gs_0 gs_1 \cdots gs_n$. In the case where the distributed computation contains 2 processes, the ith global state in the sequence is $gs_i = (s_{0,j}, s_{1,k})$, where $s_{0,j}$ represents the jth local state in the process trace for process 0. The following shows the results of a *global step* or *transition* from gs_i to gs_{i+1}, denoted $gs_i \to gs_{i+1}$.

$$
\begin{aligned}
gs_i \to gs_{i+1} \quad = \quad & (s_{0,j}, s_{1,k}) \to (s_{0,j+1}, s_{1,k}) \ \text{ OR} \\
& (s_{0,j}, s_{1,k}) \to (s_{0,j}, s_{1,k+1}) \ \text{ OR} \\
& (s_{0,j}, s_{1,k}) \to (s_{0,j+1}, s_{1,k+1})
\end{aligned}
$$

There do exist constraints that restrict certain transitions. Suppose the system is in a global state $g_i = (s_{0,i}, s_{1,j})$. If a message m has not yet been sent when process 0 is in $s_{0,i}$, and the transition $s_{1,j} \to s_{1,j+1}$ is the result of process 1 receiving the message m, then the global transition $(s_{0,i}, s_{1,j}) \to (s_{0,i}, s_{1,j+1})$ is not allowed. In other words, a process should not be permitted to receive a message before it is sent.

It is important to note that a global step, $gs_i \to gs_{i+1}$ is nondeterministic. As a result, an execution or *run* of a distributed program, which defines a particular partial ordering of local states, is not represented by a unique global trace but rather a set of global traces.

Definition 3.4 *The function* global() *is a mapping from a run, r, to a set of global traces that are possible in r.*

3.2 Distributed Predicates

Next, we present three classes of predicates defined on a global trace. These are: *local*,*non-local* and *global*.

A local predicate is satisfied strictly by the sequence of local states of a single process. For example, the predicate,

$$(x = 10 \text{ before } y = 20 \text{ in } P_0),$$

can be detected by looking at the process trace for P_0. A special subclass of the local predicates is the *simple* predicates. This is a predicate that is satisfied by a local state.

Non-local predicates are composed of one or more local predicates that are evaluated against each other using boolean or temporal operators. For example, the non-local predicate

$$(x > 10 \text{ in } P_0) \wedge (y > 5 \text{ in } P_1)$$

is satisfied if the global trace contains a global state where the local predicates are satisfied. Similarly the non-local predicate

$$(z \leq 20 \text{ in } P_0) \; before \; (w < 10 \text{ in } P_1)$$

is satisfied if there exists a global state where $(z \leq 20)$ is true and then sometime later there exists a global state where $(w < 10)$ is true.

The last class of predicates are the global predicates. Basically, these are the predicates that denote any type of global property on the global trace, for example

$$(x \text{ in } P_1) + (y \text{ in } P_2) + (z \text{ in } P_3) \geq 20.$$

The sets of predicates for each described class are not disjoint. In fact, if L, N, and G represent the set of local, non-local and global predicates respectively, then $L \subset N \subset G$.

The local, non-local and global predicates are all defined on a global trace. However, for a given run r, there does not exist a unique global trace but rather a set of global traces that are possible, which is denoted by $global(r)$. By adding a quantifier it is possible to detect the predicate in either a *strong* or *weak* sense for run r. A strong predicate is satisfied if the predicate is true in all global traces in $global(r)$, whereas a weak predicate is satisfied if the predicate is true in at least one of the global traces in $global(r)$.

In the next section, we will present Waldecker's logic for defining distributed predicates. The logic can specify weak and strong forms of the simple and some of the non-local predicates, however, it is unable to specify strong and weak forms of predicates that are exclusively in the global predicate set like the logics of Cooper[4] and Spezialetti[21] can do.

3.3 A Logic for Defining Distributed Predicates

The following is the syntax for specifying predicate formulas[24].

<u>Logic Syntax</u>

$$
\begin{array}{rcl}
<formula> & ::= & A :<linear> \mid E :<linear> \\
<linear> & ::= & \Box <bool> \mid \Diamond <bool> \mid <bool> \rightarrow <bool> \\
<bool> & ::= & <simple> \mid <bool> \wedge <bool> \mid \\
 & & <bool> \vee <bool> \mid \neg <bool> \\
<simple> & ::= & \text{predicate based on state information of single process}
\end{array}
$$

Next, we specify the semantics of the logic. The following notation is used: r designates a run, g represents a global trace, g^i designates the suffix of g from ith global state in the sequence, ie. $g_i \cdots g_n$, and $global(r)$ denotes the set of global traces for run r.

Logic Semantics

$$\begin{array}{rll}
r \models A :< linear > & \text{iff} & \forall g \,:\, g \epsilon global(r) \,:\, g^0 \models < linear > \\
r \models E :< linear > & \text{iff} & \exists g \,:\, g \epsilon global(r) \,:\, g^0 \models < linear > \\
\\
g^i \models < bool > & \text{iff} & < bool > \text{ true in global state } g_i \\
g^i \models \neg < bool > & \text{iff} & \neg g^i \models < bool > \\
g^i \models \Box < bool > & \text{iff} & \forall j \,:\, j \geq i \,:\, g^i \models < bool > \\
g^i \models \Diamond < bool > & \text{iff} & \exists j \,:\, j \geq i \,:\, g^i \models < bool > \\
g^i \models < bool_1 > \rightarrow < bool_2 > & \text{iff} & \exists j,k \,:\, i \leq j < k \,:\, g^j \models < bool_1 > \wedge \\
& & g^k \models < bool_2 >
\end{array}$$

A *formula* designates a strong or weak distributed predicate defined on a run. A *linear* is a predicate that is defined on a sequence of global states, and a *bool* is a boolean expression that is defined on a global state.

It is desirable to be able to detect any distributed predicate that can be specified by the above logic. Currently, however, the complexity of the algorithms to detect many of the predicates make it inappropriate to do so online. Bryan Chin has built an offline debugger, though, which is able to detect any predicate specified by the logic's syntax[3].

For our online debugger, we have chosen a subset of the distributed predicates to detect. The criteria used for deciding which ones to detect were complexity of the detection algorithm and usefulness of the predicate to a programmer. The names and description of the predicates chosen are presented next.

The first type of predicate which is called *weak conjunctive predicate* (WCP) is $E : \Diamond(prod)$ where *prod* denotes the conjunction of simple predicates which is a special case of the $< bool >$ predicate. WCP is true for run r if there exists a global trace, $g \,\epsilon\, global(r)$, such that *prod* is true in at least one of the global states in the global trace. We will write it as $WCP(prod)$.

WCP is a very useful predicate for detecting *prod* predicates that are indicative of an error. Suppose *prod* indicates an error global state such as two processes updating different copies of a replicated file at the same time. Even though the error may not be observed as the program is executing, if $WCP(prod)$ is detected, then the programmer knows that given a different interleaving of local states, the error will occur. Basically, WCP enables us to detect any consistent global state that can occur for a given partial ordering of local states.

We also detect $A : \Diamond(prod)$. It is called *strong conjunctive predicate* (SCP) and is true for run r if for all global traces, $g \,\epsilon\, global(r)$, there exists at least one global state in g where *prod* is true. We will write the predicate as $SCP(prod)$. This predicate is useful for specifying a *prod* that must be true for at least one global state in the run regardless of the interleaving of local states. For example, in the case of a distributed two-phase commit protocol, if the master decides to commit a transaction, then it must be true that the program was in a global state where all the slave were "ready" to commit. If the program is executed and commits but a global state where all slave processes are "ready" does not occur then the program has an error in it. With the SCP predicate this can be checked.

The last type of predicate that we explicitly detect in our debugger is the predicate $A : (chain)$, where *chain* denotes a special case of $<bool>\rightarrow<bool>$. A chain is a list of simple predicates, sp, linked with the $before$ arrow, for example, $sp_1 \rightarrow sp_2 \rightarrow sp_3$. This predicate is called *strong linked simple predicate* (SLP) and is written as $SLP(chain)$. The predicate is true for run r if for all global traces, $g \,\epsilon\, global(r)$, there exist a subsequence of global states in the global trace where the predicates are true in the order listed in the chain. This predicate is useful for detecting if a certain sequence of actions always occur for a given run r.

Finally, the debugger is also able to detect the distributed predicate $E : (chain)$ which is called *weak linked simple predicate* (WLP). This is true for run r if there exists a global trace, $g \epsilon global(r)$ such that there exists a subsequence of global states in g where the simple predicates are true in the order listed in the chain. The debugger does not explicitly detect WLP, however as is shown in [24], it can be detected using WCP in conjunction with SLP. If p and q are simple predicates,

$$E : \ p \rightarrow q \Leftrightarrow (A : \ p \rightarrow q) \vee (E : \Diamond(p \wedge q))$$

3.4 Predicate Detection

This section will present the algorithms for detecting the distributed predicates WCP, SCP and SLP. Before doing so, it is necessary to preface the discussion by presenting vector time which is a logical time that can be used to infer the partial ordering of the local states in a distributed computation[13].

Vector time is represented by an n-dimensional vector. Each process, P_i, maintains a vector clock, $vc_i[1..n]$ where n is the number of processes. The element $vc_i[i]$ represents the current logical time of P_i. The element $vc_i[j]$ where $i \neq j$, represents that latest event in P_j that P_i knows about.

After P_i assigns a timestamp to an event, $vc_i[i] = vc_i[i] + d$ where $d > 0$. The vector clock, $vc_i[1..n]$ is appended to every message leaving P_i. When P_i receives a message, it strips the vector clock, $in\_vc[1..n]$, from the incoming message and adjusts its vector in the following way:

$$1 \leq k \leq n \; : \; vc_i[k] ::= max(vc_i[k], in\_vc[k])$$

Timestamps can be used to determine if events are ordered. Suppose event p in P_i has timestamp ts_i and event q in P_j has timestamp ts_j.

$$p < q \Leftrightarrow ts_i[i] \leq ts_j[i]$$

To detect if two events are *unordered* or *concurrent*, $p \parallel q$,

$$p \parallel q \Leftrightarrow (ts_i[i] > ts_j[i]) \wedge (ts_j[j] > ts_i[j])$$

The WCP and SCP predicates are detected by accumulating at a checker process timestamped state information from the application processes. The checker process recreates *possible* global states to determine if WCP or SCP were true. In the case of an SLP predicate, such as $SLP : sp_1 \rightarrow sp_2$, the predicate is detected if and only if there exists a causal relation between the occurrences of sp_1 and sp_2.

3.4.1 Detecting Weak Conjunctive Predicate

First, we will present a necessary and sufficient condition for the $WCP(prod)$ to hold. Let *prod* consist of the conjunction of simple predicates, $p_1 \wedge p_2 \wedge \cdots \wedge p_n$, where p_i is in process P_i. Let $p_i(s)$ denote the state of predicate p_i in local state s.

Theorem 3.1 *$WCP(prod)$ is true iff $\forall i : 1 \leq n \; : \; \exists s_i \epsilon P_i$ such that $p_i(s_i) = true \; \wedge \; \forall j, k \; : \; j \neq k \; : \; s_j \parallel s_k$*

Proof: see [24] □

We assume that there exists a monitor for each process. When a monitor detects that a simple predicate in *prod* is *true*, the monitor timestamps the local state and sends the timestamp to a central checker. To detect WCP, the checker tries to find, for each simple predicate p_i, a timestamp, ts_i, of a local state where p_i is true such that $\forall i, j \ : \ i \neq j \ : \neg(ts_i < ts_j)$. In other words, the checker must find an antichain of size n to detect $WCP(prod)$.

The pseudo-code for the checker process is in Figure 3.1. The checker maintains a separate queue for each simple predicate p_i where it puts incoming timestamps corresponding to local states where p_i is true in P_i. For the algorithm to work correctly, it is important that the timestamps from a monitor to the checker be in a FIFO order. This can easily be ensured by keeping sequence numbers. Whenever the checker receives a timestamp, the function in Figure 3.1 is run which returns a boolean value indicating if WCP has been detected.

The algorithm works by comparing heads of queues. If none of the n queues are empty and timestamps at the head of the queues are incomparable then $WCP(prod)$ is $true$. There are three loops in the algorithm. The inner two loops are used to compare the head of the queues against each others. Let $head_i$ denote the timestamp at the head of p_i's queue. If $\exists i, j \ : i \neq j \ : \ head_i < head_j$ then the index i is placed in a set called $newlow$. After the inner two loops have completed, the set low is set equal to $newlow$ and the timestamps at the head of the queues whose indices are in low are discarded. It is valid to do this because if $head_i < head_j$ and the timestamps in p_j's queue are totally ordered, then $head_i$ "happened before" all the timestamps in p_j's queue after $head_j$. Therefore, $head_i$ cannot be part of an antichain of size n so it can be safely discarded. If *low* is empty after the two inner loops finish, the outer loop exits. At this point, if none of the queues are empty, $WCP(prod)$ is *true*.

```
VAR  q1,q2,...,qn: queues

/* function returns boolean indicating if WCP detected */

function findWCP(ts,i)  /* input:  timestamp from Pi */
{
Const   all={1,2,...,n};
VAR     low,newlo : subsets of all;
        j : 1...n;

if notEmpty(qi) then insert(qi,ts)
else
   {
   insert(qi,ts);
   low = {i};
   while (low != EMPTY)
      {
      for i in low
         {
         for j in all
            {
            if  head(qi) < head(qj) then union(newlo,i);
            if  head(qj) < head(qi) then union(newlo,j);
            }
         }
      low = newlo;
      for i in low
         {
         deletehead(qi);
         }
      }
   if none of the queues are empty, return true
   }

return false

}
```

Figure 3.1: Pseudo-code for WCP checker process

Lemma 3.1 shows that to guarantee the algorithm will detect the occurrence of WCP, the monitor watching p_i in P_i only needs to detect the first local state where p_i is *true* and the first local state where p_i is *true* after each send event in P_i.

Lemma 3.1 *Let $WCP(p_1 \wedge p_2)$ be true for a run r, where p_1 is in P_1 and p_2 is in P_2. If the monitor watching p_i sends timestamps to checker process the first time p_i is true and the first time p_i is true after each send event then $WCP(p_1 \wedge p_2)$ will be detected by our checker algorithm.*

Proof: Suppose $WCP(prod)$ cannot be detected by sending timestamps the first time a simple predicate is true and the first time a simple predicate is true after each send event. Looking at Figure 3.2, s_1 is a local state where p_1 first occurs after a send, and s_2 is a local state after s_1 and before the next send event where p_1 also occurs.

Let us suppose that if the timestamp for s_2 is sent then $WCP(prod)$ is detected and that global state (s_2, s_j) is the global state where $p_1 \wedge p_2$ is true. Therefore, for the global state (s_1, s_j) not to detect WCP means that the following must be true.

$$((s_j < s_1) \wedge (s_2 \parallel s_j)) \vee ((s_1 < s_j) \wedge (s_2 \parallel s_j))$$

However, if $(s_j < s_1)$ then we know that

$$(s_j < s_1) \wedge (s_1 < s_2) \Rightarrow (s_j < s_2), \text{ a contradiction.}$$

Additionally, if $s_1 < s_j$ and there is no send event between s_1 and s_2 then $s_2 < s_j$, a contradiction. Thus, it must be true that s_1 and s_j are incomparable and can detect $WCP(prod)$. We conclude that if a monitor sends a timestamp the first time a predicate is true and the first time a predicate is true after each send event then WCP will be detected if it occurs. $\square$

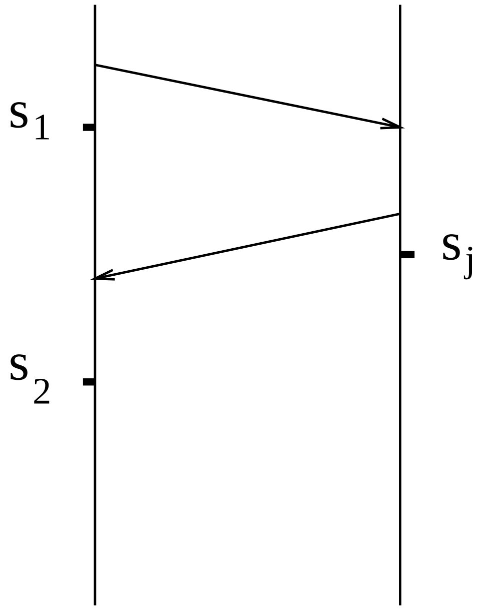

Figure 3.2: WCP lemma.

3.4.2 Detecting Strong Conjunctive Predicate

First, we will define the notion of an interval within a process trace and then present a necessary and sufficient condition for $SCP(prod)$ to hold.

Definition 3.5 *An* interval*, I, consists of a consecutive sequence of local states from a process trace with $lo(I)$ designating the* start *state and $hi(I)$ designating the* stop *state of the interval.*

Definition 3.6 *A* simple predicate interval *is an interval where a simple predicate p_i is true. It is denoted by $SPI(p_i)$.*

Definition 3.7 *A set of intervals, $I_1, I_2, \cdots, I_n$* overlap *if and only if the lo's of all the intervals "happened before" the hi's of all the intervals.*

$$overlap(I_1, I_2, \cdots, I_n) \Leftrightarrow \forall i, j \; lo(I_i) < hi(I_j)$$

A necessary and sufficient condition for $SCP(prod)$ to hold is presented in theorem 3.2.

Theorem 3.2

$$SCP(p_1 \wedge p_2 \wedge \cdots \wedge p_n)$$

$$\Leftrightarrow$$

$$overlap(SPI(p_1), SPI(p_2), \cdots, SPI(p_n))$$

Proof: see [25]. □

We assume that there exists a monitor for each process. When a monitor detects a simple predicate interval, the two timestamps corresponding to the lo and hi of the interval are sent to a centralized checker process. To detect SCP, the checker tries to find a simple predicate interval for each p_i such that the set of intervals overlap.

The pseudo-code for the checker process is shown in Figure 3.3. The code is essentially identical to the pseudo code for the WCP checker in Figure 3.1 with the exception that the innermost loop is comparing two intervals to see if they overlap as opposed to checking two local states to see if they are concurrent.

It is possible for the checker to detect $SCP(prod)$ without requiring the monitors to send every simple predicate interval they detect to the checker. Lemma 3.2 presents the rules for deciding which simple predicate intervals to send and shows that these intervals are guaranteed to detect SCP if it occurs. The following definition is used in the lemma.

```
VAR  q1,q2,...,qn: queues;

/* function returns boolean indicating if SCP detected */

function  findSCP(I,i)  /* input:  interval I  from Pi */
{
Const   all={1,2,...,n};
VAR     low,newlo : subsets of all;
        j : 1...n;


if notEmpty(qi)  then
   {
   insert(qi,I);
   }
else
   {
   insert(qi,I);
   low = {i};
   while (low != EMPTY)
      {
      for i in low
         {
         for j in all
            {
            if  head(qi).hi < head(qj).lo then union(newlo,i);
            if  head(qj).hi < head(qi).lo then union(newlo,j);
            }
         }
      low = newlo;
      for  i in low
         {
         deletehead(qi);
         }
      }
   if none of the queues are empty, return true;
   }

return false;
}
```

Figure 3.3: Pseudo-code for SCP checker process

Definition 3.8 *An interval ,I, comes after the local state s if* $s < hi(I)$.

Lemma 3.2 *Let* $SCP(p_1 \wedge p_2)$ *be true for run* r*, where* p_1 *is in* P_1 *and* p_2 *is in* P_2*. If the monitor watching for the simple predicate interval where* p_i *is true sends timestamps to the checker for the first interval it detects and for the first interval after each receive event in* P_i *then* $SCP(p_1 \wedge p_2)$ *will be detected by our checker algorithm.*

Proof: Suppose $SCP(prod)$ cannot be detected by sending the first interval and first interval after each receive event to the checker. Looking at Figure 3.4, I_1 is the first simple predicate interval after a receive event, and I_2 is a simple predicate interval after I_1 and before the next receive event.

Let us suppose that if the interval I_2 is sent then $SCP(prod)$ is detected by interval pair (I_2, I_j). Therefore for the interval pair (I_1, I_j) not to detect the SCP, the following must be true.

$$\neg(lo(I_1) < hi(I_j)) \quad \vee \quad \neg(lo(I_j) < hi(I_1))$$

However, if $(lo(I_2) < hi(I_j))$ then we know that,

$$(lo(I_1) < lo(I_2)) \wedge (lo(I_2) < hi(I_j)) \Rightarrow (lo(I_1) < hi(I_j)), \text{ a contradiction.}$$

Additionally, if $(lo(I_j) < hi(I_2))$ and there is no receive event between $hi(I_1)$ and $hi(I_2)$ then $(lo(I_j) < hi(I_1))$, a contradiction. Thus the pair (I_1, I_j) must overlap and detect SCP. We conclude that if the first interval is sent to the checker and first interval after each receive event is sent to the checker, then SCP will be detected if it occurs. □

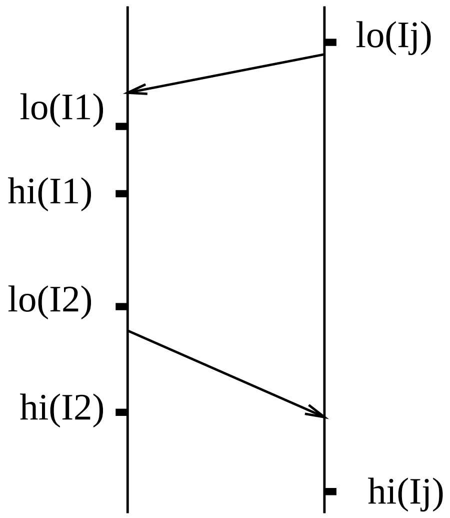

Figure 3.4: SCP lemma.

3.4.3 Detecting Strong Linked Simple Predicate

For $SLP(p_1 \rightarrow \cdots \rightarrow p_n)$ to be satisfied in run r, the simple predicates must be true in the order listed for every global trace, $g\epsilon global(r)$. For the chain $(p_1 \rightarrow p_2 \rightarrow p_3)$, where p_1,p_2 and p_3 are true in s_1, s_2 and s_3 respectively, the only way to guarantee that the local states occur in the order s_1, s_2 then s_3 is if $s_1 < s_2 < s3$. Therefore, if we can detect a set of appropriately ordered local states where the simple predicates are true, we can guarantee that the SLP predicate is $true$. In [24], Waldecker shows how this is also a sufficient condition for detecting the SLP predicate.

The algorithm to detect the predicate $SLP(p_1 \rightarrow \cdots \rightarrow p_n)$ works as follow. Each process maintains a variable called $rightmost$. A process assumes that all the simple predicates up to $rightmost$ have been detected in the correct causal order. Initially, $rightmost$ is initialized to 1 at all processes, so only the monitor for the first simple predicate, p_1, will be active. When the process detects p_1, it increments its variable *rightmost* by 1.

Each process appends the variable *rightmost* to every outgoing message. When a process receives a message, it strips the variable *temp_rightmost* from the message and adjusts its *rightmost* variable in the following way,

$$rightmost = max(rightmost, temp\_rightmost).$$

When the variable changes, the process checks to see if it needs to the detect the next simple predicate in the chain. Once a process detects a simple predicate, it increments *rightmost* by one, unless the simple predicate is the last on in *chain*, in which case, the SLP predicate is *true*.

In this chapter we have presented distributed predicates. and have described the types of distributed predicates that can be detected by the online debugger we have built. We have also outlined the distributed algorithems used to detect these distributed predicates. Two of the algorithms are assymetric and use a checker process to recreate global state using vector time.

Chapter 4

Online Debugger

This chapter discusses the design and implementation of the online distributed debugger along with the environment for creating event-driven distributed programs.

4.1 Distributed Programming Environment

In this section, we present the environment used to create event-driven distributed programs. The online debugger we have designed and built is able to debug distributed programs written in this environment. The programs are written in C and utilize the SunOs UNIX operating system.

The "events" that drive the computation of a distributed program are messages and keyboard input. When a process receives an event, it executes a routine to handle the event type. When it finishes the handling routine, the program reenters a wait loop until the next event arrives. An example of a more sophisticated distributed programming environment that has a similar event-driven structure is ISIS[26].

Assuming no reference is made to real-time in the event handling routines, the sequence of actions in the handling routines are deterministic. However, the behavior of the distributed computation as a whole is nondeterministic. This is because factors such as network load, cpu load and peripheral device load (such as disk drives) can change the relative ordering of actions in the processes and thus affect the order that messages are received at a process.

A distributed computation involves the cooperation and communication of several processes. Each process is written as an event-driven program consisting of two major parts: the controller and the application.

In general, the controller portion is just a busy-wait loop. It waits to receive messages from other processes or input from the keyboard. When a message is received (keyboard input is considered to be a message of type STD_IN), the controller examines the message type. If the message is a special predefined type, then the controller processes the message, but if the message is a user-defined type, then the controller passes control to a handler-routine in the application portion of the program. Once the handler-routine has finished, control passes back to the controller. The main purpose of the controller is to enforce the event-driven policy while the application portion carries out the logistics of the distributed application. The controller part does not change from one application to the next like the application part does.

The processes in a distributed computation are not dynamically created. Each process must be explicitly started by the programmer. Once created, the programmer must explicitly establish a logical message channel between two processes before they can directly communicate.

To create a distributed program the user writes the application part. An example of the application portion of a distributed program whose objective is to respond to a $PING$ message by sending with a $PONG$ message to the source is shown in Figure 4.1.

```
#include "application.h"
#define   PING   50
#define   PONG   51

extern  Proc  *peers;  /* list of neighbor processes */
extern  int    myId;   /* logical id of the process  */

/*--------------------------------------------*/
/*  Global Application Variable               */
/*--------------------------------------------*/
int   whoSentPing;


void   processEvents(msg,event,source)
   char   *msg;
   int     event;
   int     source;
{
int   whoToPing;

switch(event)
   {
   case   PING:
      {
      whoSentPing=source;
      printf("Received PING from %d\n",whoSentPing);
      sendMess(whoSentPing,PONG,NULL,NULL);
      break;
      }
   case   PONG:
      {
      printf("Received PONG from %d\n",source);
      break;
      }
   case   STDIN:   /* predefined type */
      {
      if (msg[0] == 'p')
         {
         get application process to PING from message;
         sendMess(whoToPing,PING,NULL,NULL);
         break;
         }
      }
   }
} /* end processEvents() */
```

Figure 4.1: Example of an event-driven program

The user must first define the message types for the distributed computation. In this case, two types are defined; they are $PING$ and $PONG$. Next, the user must write the function $processEvents()$ as is shown in Figure 4.1. Whenever the controller receives a message whose type it cannot handle, it passes control to $processEvents()$ to see if the user has written a routine to handle the message. The input parameters to $processEvents()$ are the following: msg, $event$ and $source$. The parameter msg is a pointer to a buffer containing the message data, whereas $event$ is the message event type. The logical id of the process that sent the message is contained in the variable $source$.

This example in Figure 4.1 illustrates the use of the function $sendMess()$ which is the non-blocking communication primitive used to send messages between processes linked with a communication channel. Asynchronous messages are the only type that can be sent. The function $sendMess()$ has four input parameters: $logId$, $type$, msg and len. The parameter $logId$ is the logical id of the destination process. The parameter $type$ is the message type being sent. The parameter msg is a pointer to a buffer containing the message data, and the length of the message data is len. In the $PING$ handling routine in Figure 4.1, $sendMess()$ is used to send a $PONG$ message. In this case, no data is being sent in the message so the msg and len parameters are set to $NULL$.

In summary, the distributed programming environment was created so that it would be relatively easy to develop simple distributed programs. These distributed programs can be used to test and evaluate distributed algorithms. The programs are also used as target programs for our online debugger.

4.2 Online Debugger

The online debugger is able to detect global states or sequences of global states in a distributed computation. The three types of distributed predicates that the online debugger can explicitly detect are *strong conjunctive predicate* (SCP), *weak conjunctive predicate* (WCP) and *strong linked predicate* (SLP).

Like the distributed programs it debugs, the online debugger is an event-driven distributed program. The debugger is divided into two major parts which correspond directly to the two types of processes in the system. They are the *coordinator* part and the *monitor* part which are shown in Figure 4.2. There exists only one coordinator process, but the number of monitor processes is the same as the number of *application* processes. Application processes are the processes in the distributed computation being debugged. Each application process is paired with a monitor process as illustrated in Figure 4.2. The major roles of the coordinator and monitor are discussed below.

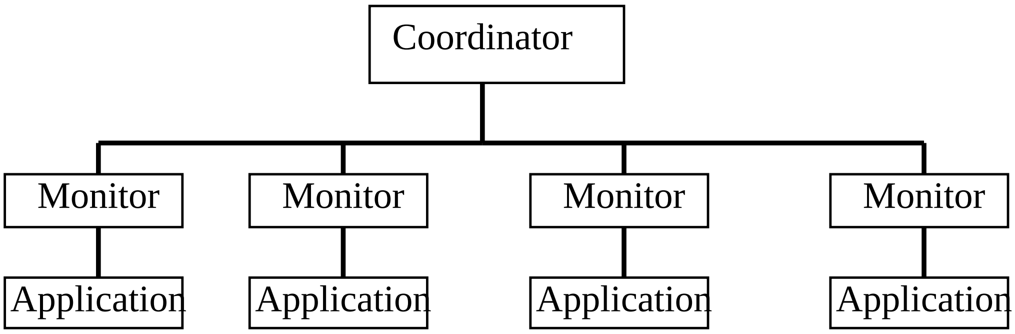

Figure 4.2: Process structure of our online debugger

Coordinator:

- To monitor the status of the distributed computation. For example, it detects when application processes are created or terminated.
- To accept input from programmer detailing which distributed predicates should be detected.
- To detect distributed predicates WCP and SCP by collecting and analyzing the simple predicate information from the distributed monitor processes.
- To report to the programmer the detection of WCP and SCP predicates in the distributed computation.
- When WCP is detected, to compute a consistent global state where the distributed predicate is $true$ and to send this global state information to the monitor processes which rollback the distributed computation.

Monitor

- To detect simple predicates defined within the domain of the application process it is monitoring.
- To cooperate with the other monitor processes to detect SLP.
- To screen and modify messages to or from the application process it is monitoring.

- To rollback the application process after WCP has been detected at the coordinator. This restores the distributed computation to a consistent global state where the distributed predicate occurred.

Those are the major roles of the two types of processes that makeup the online distributed debugger.

4.2.1 How Debugger Operates

This section will provide a general overview of the online debugger's operation.

Inputting Distributed Predicates: The user inputs the distributed predicates to be detected at the coordinator. The coordinator maintains a list of the distributed predicates that have been input. An example of a distributed predicate is the following,

$$WCP\text{::}\ P_1\!:\!(x > 10) \wedge P_2\!:\!(y > 5).$$

As the distributed predicates are input, they are assigned unique predicate identification numbers and put into the distributed predicate list. Every distributed predicate is expressed in terms of simple predicates or local predicates. These are predicates whose domain is the state space of an individual process. In the above example, the WCP predicate is built from the simple predicate $(x > 10)$ in process P_1 and simple predicate $(y > 5)$ in process P_2. After a distributed predicates has been entered, the coordinator sends messages to the monitors telling them which simple predicate to detect.

Monitors Detecting Simple Predicates: Each monitor process maintains a list of simple predicates it is trying to detect. The way that simple predicates are detected is relatively simple. The monitor *singlesteps* the application process. When the application has finished stepping, the monitor examines the address space of the application process to see if any of the simple predicates in its predicate list are $true$.

Every simple predicate in a monitor's predicate list is part of a distributed predicate of type WCP, SCP or SLP. When the monitor detects a simple predicate, the resulting action depends upon the type of distributed predicate the simple predicate is a part of.

When a detected simple predicate is part of a WCP predicate, the event is given a vector clock timestamp. Next, the timestamp along with the unique identifier for the distributed predicate is sent to the coordinator for analysis. Note however, as we have shown in lemma 3.1, it is sufficient for the monitor to detect the first occurrence of the simple predicate and the first occurrence of the simple predicate after the application process has sent a message. The simple predicate can be ignored by the monitor at all other times.

For a simple predicate that is part of an SCP predicate, the monitor is trying to detect an interval where the simple predicate is true. Therefore, the monitor watches for transition of the predicate from $false$ to $true$ and then from $true$ to $false$. Once an interval has been detect, the monitor takes the timestamps corresponding to the start and end of the interval respectively and sends them to the coordinator for analysis. However, as was shown in

Lemma 3.2, there exist cases where the monitor can deduce that the starting and ending timestamps will not provide the coordinator with any useful information in detecting the SCP predicate. In these cases the monitor can discard the information instead of sending it to the coordinator.

Whereas the WCP and SCP predicates are detected by the coordinator, this is not true for an SLP predicate. An SLP predicate is detected by a monitor process. The general form of an SLP predicate is the following,

$$SLP :: sp_1 \rightarrow sp_2 \rightarrow sp_3 \rightarrow sp_4 \rightarrow \cdots \rightarrow sp_n,$$

where sp_i is a simple predicate.

Each monitor maintains a status vector that reflects the monitor's knowledge of the rightmost simple predicate that has been detected in the correct causal order for each SLP predicate. The monitor appends the status vector to every message leaving the process it is monitoring. Before relaying an incoming message to its application process, the monitor strips the status vector from the incoming message and adjusts its own status vector by taking a component-wise maximum of the incoming status vector and its current status vector.

The SLP predicate is $true$ when the chain of simple predicates has been detected. The monitor that detects the rightmost simple predicate in the chain can immediately notify the user that the SLP predicate is $true$. This is in contrast to the WCP and SCP predicates where there can exist a substantial delay between the occurrence and detection of the predicate.

Coordinator Detecting Distributed Predicates: The coordinator maintains a collection of queues for each SCP and WCP expression, one for each simple predicate in the expression. The queues are used to store timestamps that indicate when the simple predicates were true. When a simple predicate that is part of a WCP predicate is true, the monitor timestamps the event and sends the timestamp to the coordinator. Upon receiving the timestamp, the coordinator places it in a queue and tries to detect the WCP predicate by running the WCP detection algorithm. For a predicate that is part of a SCP predicate, the interval it was true is timestamped by a monitor and sent to the coordinator. When the coordinator receives the interval, it places the interval in a queue and runs the SCP detection algorithm to see if the SCP predicate has occurred. The centralized detection algorithms for SCP and WCP predicates are described in chapter 3.

Restart After WCP Detection: One reason for using a WCP predicate is to detect global states that indicate an error. This is because WCP can detect any consistent global state that can occur in a run. While the program executes, the global state may not actually occur, but if WCP is detected, the programmer knows that there exists an interleaving of local states such that the global state could have occurred.

Given that a WCP predicate is used to find a global state that is indicative of an error, the programmer may want to analyze the processes in the global state where the WCP predicate is true to determine the potential cause of the error. However, the delay between a predicate's occurrence and detection can be substantial, so when the program is finally halted, it may no longer be in a state the programmer is interested in examining. Therefore, for the distributed predicate $WCP(prod)$, we give the programmer the option of rolling back the distributed computation to a consistent global state where $prod$ is $true$ once $WCP(prod)$ has been detected. The coordinator uses the set of timestamps that detected the WCP predicate to calculate this global state which it then sends to all the monitors. As the application processes are executing, they are recording incoming events to a file. So, when a monitor receives a message telling it to roll back an application process, the monitor restarts the application process and replays the recorded events until the process reaches a local state that is part of the global state where $prod$ is $true$. This is a valid way to rollback the application processes because the event-handling routines in the application are deterministic. Eventually, when we have a more elaborate checkpointing and state recovery mechanism[22], it may not be necessary to replay all the messages. Instead, the monitor can restore the most recent checkpoint state before the desired state and then replay the logged messages from the checkpoint state to the desired state.

4.2.2 Design Decisions

This section will discuss in more detail certain aspects of the online debugger's design.

System Messages versus Program Messages: The message channels in the debugger system can be divided into two types. The first type is the *debug message channels*. These channels link the monitor processes and link the monitor processes to the coordinator. The second type is the *application* *message channels* which link the monitor processes to the application processes. See Figure 4.2.

The messages that flow in the debug message channel are classified as either $SYSTEM$ messages or $PROGRAM$ messages. In general, the $SYSTEM$ messages are not part of the distributed computation being debugged but rather are used by the debugger to carry out its internal operations. For example, when the monitor detects a simple predicate, the event is timestamped and sent to the coordinator in a $SYSTEM$ message.

The $PROGRAM$ messages are part of the distributed computation. These are messages that originated from an application process and then were relayed by a monitor process onto a debug message channel enroute to a destination application process. Since these messages are part of the distributed computation, they are used to define the partial ordering of the local states in the distributed computation. The $SYSTEM$ messages are not used to infer ordering information about local states. Some debuggers, such as the one by Miller and Choi, use $SYSTEM$ messages to order local states[15]. We believe this is inappropriate because it can lead the programmer to draw conclusions about the program that are not valid when the program is not being debugged.

The format of the two types of messages is shown in Figure 4.3. The $PROGRAM$ message contains all the fields found in a $SYSTEM$ message plus some additional fields that contain vector clock information and SLP status information. The vector information is used to maintain the vector clocks. The SLP status information is used by the SLP detection algorithm.

The second type of message channels are the application message

Total Len	PROGRAM	Event ID	SLP status info	Vector Clock	Msg. Data Len	Msg. Data

a)

Total Len	SYSTEM	Event ID	Msg. Data

b)

Figure 4.3: Field format for a) PROGRAM message b) SYSTEM message. These are type of messages sent over *debug message channel.*

channels which are bidirectional channels connecting an application process to its monitoring process. These channel carry the messages that define the partial ordering of local states in the distributed computation. The field format is shown in Figure 4.4. When an application process, P_0, wants to send a message to another application process, P_1, the message is first transmitted by P_0 on its application message channel to P_0's monitor. The message is then relayed on a debug message channel as a $PROGRAM$ message to P_1's monitor who then relays it to P_1 on an application message channel.

Total Len	Source or Destination ID	Event ID	Vector Clock	Msg. Data Len	Msg. Data

Figure 4.4: Field format for messages on *application message channels*

Monitoring Simple Predicates: For the distributed predicate detection algorithms to work correctly, we must be able to detect the occurrence of simple predicates. Since we do not have access to special monitoring hardware, all of our monitoring is done in software by monitoring processes running on the same processor as the application process being debugged. Unfortunately, this makes the monitors very *intrusive* and thus the probe effect is significant.

A software monitor works by "singlestepping" the application process and checking the status of a simple predicate after each step. It checks the status of a simple predicate by examining the address space of the process. We tackled the problem of monitoring from two different approaches which are described next. At first, we used the *ptrace()* system call. This is a special debugging command in Unix that allows a parent process to control the execution of a child process. Figure 4.5 shows the ptrace connection between the monitor and application process. Using the ptrace command, the monitor singlesteps every machine instruction in the application process. It then utilizes the ptrace command to examine the address space to check the status of the simple predicates. While this approach enabled us to detect simple predicates, it was quickly abandoned because it was too slow. See Table 4.1. The manual for the Gnu gdb debugger states that the program will run approximately two orders of magnitude slower when using ptrace to detect a simple predicate. This was proven true from our experience.

Using ptrace was so slow because Unix had to make two context switches for every single instruction executed in the application process, one when the monitor issued a "singlestep" command and one to return control to the monitor after the step. Context switches can be very expensive because they generally consist of hundreds of instructions. We decided that we could substantially reduce the cost of checking simple predicates if we singlestepped

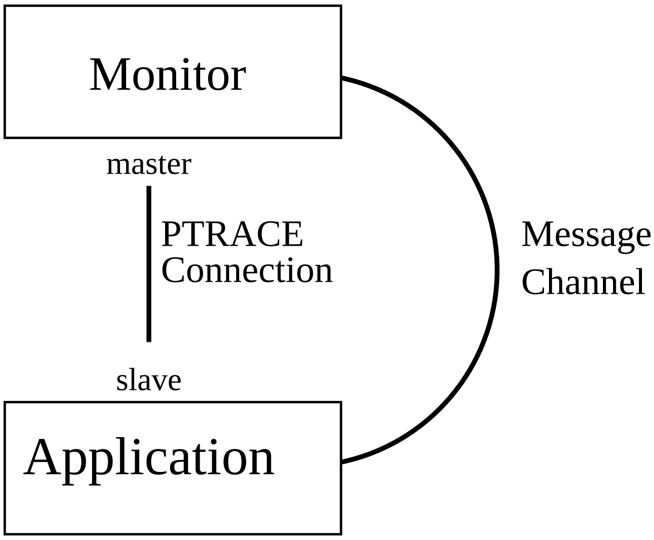

Figure 4.5: Using ptrace to monitor an application process

a C source code statement as opposed to a machine instruction. Therefore, we quit using ptrace and began using the Gnu gdb debugger in conjunction with our monitor to detect simple predicates. Using gdb enables us to take advantage of the well-developed and well-understood algorithms for single-stepping source lines that are part of the gdb debugger. The setup between the monitor, gdb and application process is shown in Figure 4.6.

A pseudo-terminal connection is established between the monitor and gdb. By using pseudo-terminal, all the standard input and standard output for the gdb debugger is from and to the monitor over the pseudo-terminal connection. By using this type of interface, we did not have to modify the code of gdb. The monitor sends gdb commands such as "step" over the pseudo-terminal connection. Once gdb indicates it is finished stepping, the monitor sends commands to gdb to check the status of simple predicates.

Using gdb turns out to be much faster than using ptrace because

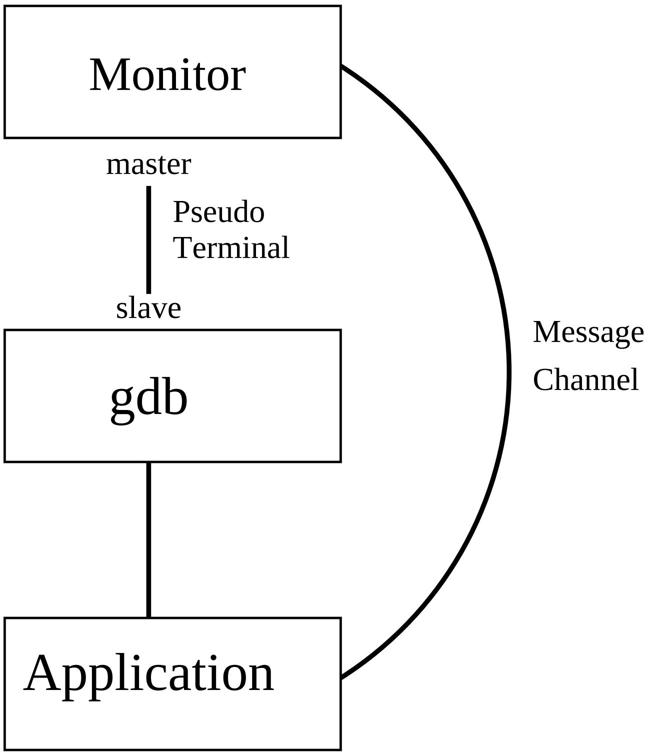

Figure 4.6: Using gdb to monitor an application process

the application is stopped less often and the status of predicates is checked less often. However, for our detection algorithms to work correctly, we must assume that a simple predicate will not change state twice in one line of C source code. We realize this assumption is not always valid, but the cases where this occur are infrequent enough so that this will not be a problem especially for a prototype debugger like ours.

There exists one restriction regarding the type of variables that can be used in simple predicates being monitored; only *global variables* can be used in simple predicate expressions. The reason for this is that we require the variable to be in existence for the lifetime of the process. With slight modifications to the debugger's code, we could also allow expressions to include *static* and *automatic* variables.

As mention in section 4.1 an application process consists of a controller part and an application part. The application part is the only part that changes from one application to the next, so the programmer is only interested in simple predicates that change state as a result of actions in the application part. Therefore, we have set up the monitor to do *selective tracing*, which means the monitor only singlesteps a process when it is in the application part of the program. This helps to speed things up and makes the system more usable especially for demonstration purposes.

Table 4.1 details the cost of monitoring a process to detect simple predicates. The data was obtained by examining the execution time of the two-phase commit protocol in the program shown in Appendix B on a Sun 3/80. When the program is not being monitored, the program takes approximately 6 seconds to execute. The other entries in the table represent the times it takes the program to execute when the debugger uses different monitoring strategies and tries to detect different numbers of simple predicates.

The ptrace command was used in two different ways which correspond to method 1 and method 2 in the table. When the monitor issues a singlestep call using ptrace, it can either wait for the application process to finish stepping or it can try to do useful work, such as receiving incoming messages from other monitor processes, while the application process is waiting to be stepped by the operating system. The techniques correspond to method 1 and method 2 in the table respectively. Initially, we were just using method 1 and this approach was quickly dropped because it was unbearably slow. We did not go back to look at method 2 until after we had finished building the debugger using gdb. We are unable to account for the dramatic difference in execution times between method 1 and method 2 when using ptrace.

The cost incurred to watch an application program can be split into two components; time to step and the time it takes to examine the state of a simple predicate. When gdb was used to trace the application "at all times," the time to step was approximately 30 seconds which increased the execution time by a factor of 6. The execution time increased by an additional 30 seconds for each additional predicate the monitor tried to detect.

When the monitor used gdb to "selectively trace," the time to step was approximately doubled. It doubled again for each additional predicate the monitor tried to detect. This penalty paid when using "selective trace" will vary depending upon the percentage of time the application process spends in

Monitoring Strategy	No. Preds. to detect	Time (in seconds)
No Monitoring	0	6
step with ptrace -Method 1	0	1800
step with ptrace -Method 2	0	100
step with gdb	1	63
step with gdb	2	93
step with gdb	3	133
selective step with gdb	1	20
selective step with gdb	2	26
selective step with gdb	3	32

Table 4.1: The cost of monitoring a distributed program

the application portion versus the controller portion of a program.

Chapter 5

Distributed Application Examples

This chapter presents case examples of event-driven distributed programs that have been written and shows how the online debugger can be used to debug the programs. Three application examples are presented. The first is a distributed mutual exclusion program, the second is a distributed two-phase commit protocol and the last example is a distributed leadership election program.

5.1 Distributed Mutual Exclusion Program

A requirement of a distributed mutual exclusion program is to ensure that at most one process is in its critical section (CS) at any given time. This is what defines a "correct" mutual exclusion protocol. Some additional concerns are to prevent starvation and deadlock as well as to ensure fairness. For the distributed mutual exclusion algorithm that we present, we will only be be concerned with showing that correctness is not violated.

The algorithm we have implemented, which is shown in Appendix A, is Ricart and Aggrawala's mutual exclusion algorithm[18]. It works as follows. Each process P_i maintains a clock called CS_clock. When a process wants to enter its CS, it timestamps the request by setting $request\_ts = CS\_clock$ and then sends the tuple $(request\_ts, i)$ to each process using a $REQUEST$

message. The process receives approval to enter its CS once it has received a $REPLY$ message from each process.

When a process P_j receives a $REQUEST$ tuple $(in\_request\_ts, i)$ from P_i, it adjusts its CS_clock to $max(CS\_clock, in\_request\_ts + 1)$. Next, one of three things happens. First, if P_j does not want to enter its CS, then it immediately sends a $REPLY$ message to P_i. If P_j is waiting approval to enter its CS, then it compares the incoming tuple with the one it has sent out. If the following is true

$$(in\_request\_ts < request_ts) \vee ((in\_request\_ts = request_ts) \wedge (i < j))$$

then P_j immediately sends a $REPLY$ message to P_i, else it places the incoming request into a $wait$ queue. After a process has finished executing in its CS, it takes all tuple requests in its $wait$ queue and sends a $REPLY$ message for each one.

The mutual exclusion program is shown in Appendix A. To demonstrate how the WCP predicate can be used, we have put an error in the $if - then$ statement at NOTE 1. When comparing timestamps, we have used a $\leq$ instead of a $<$ which will allow more than one process into its CS at the same time. Using incorrect relational operators in $if - then$ statements is actually a very typical type of program error. Figure 5.1 shows the space-time diagrams for a correct and incorrect mutual exclusion program.

Just before a process enters its CS, it set its non-shared variable $inCS$ to 1 and then sets $inCS$ to 0 upon exiting its CS. This enables us to check to see if it is possible for two processes to be in the CS at the same time for a

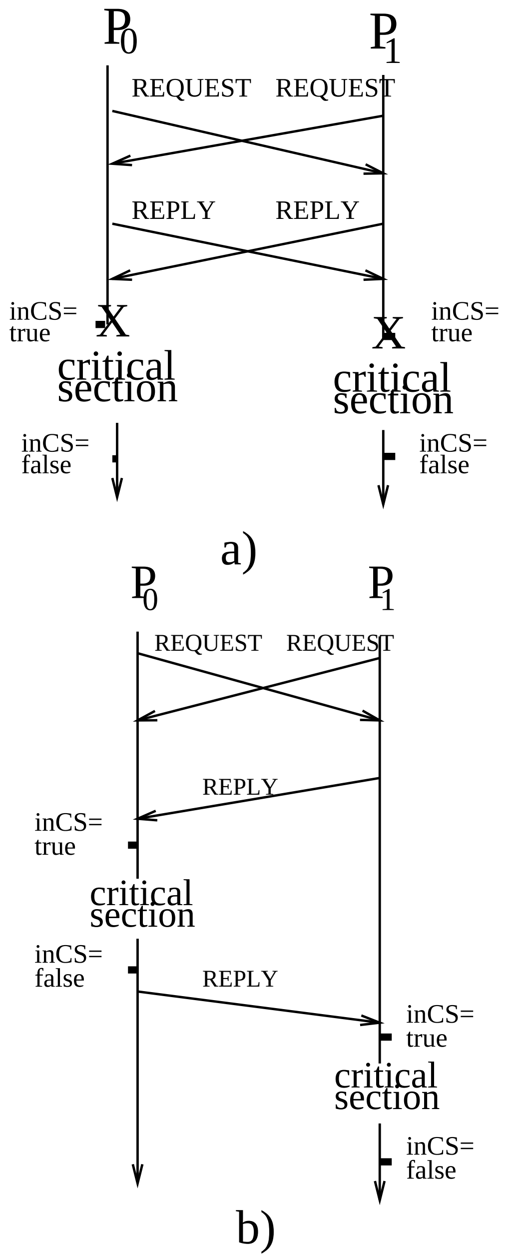

Figure 5.1: Mutual exclusion a) violated b) not violated

given run using the following WCP predicate.

$$WCP: \ (inCS == 1 \text{ in } P_0) \wedge (inCS == 1 \text{ in } P_1)$$

Additionally, by using the rollback halting algorithm, when the above predicate is detected, the program is halted in a consistent global state where

$$(inCS == 1 \text{ in } P_0) \wedge (inCS == 1 \text{ in } P_1)$$

is true. In the case of a) in Figure 5.1, upon detection of the WCP predicate, the program is halted in the global state denoted by the X's.

5.2 Distributed Two-Phase Commit Protocol

Transactions are composed of a sequence of actions. A transaction should be atomic in that either "all or none" of the actions occur. This is achieved by ensuring that all the actions can be undone until the transaction is finally *committed*. When all the actions of a transaction are local to a process, the commit procedure is relatively straightforward, however, this is not the case for a distributed transaction where the actions are distributed among the processes in a distributed system. One method of ensuring the atomicity of a distributed transaction is to use a *two-phase commit protocol*.

For a distributed transaction utilizing a two-phase commit protocol, there exists a *master* process and *slave* processes. The master oversees the transaction and makes sure that all the slaves commit or all the slaves abort. When the master is ready to commit, it initiates the first phase by sending a $PREPARE$ message to all slaves. If a slave is ready to commit when it receives a $PREPARE$ message then it replies with a $READY$ message else it replies with an $ABORT$ message. The second phase is started when either the master receives an $ABORT$ message from a slave or receives a $READY$ message from all the slaves. In the former case, the master aborts the distributed transaction by sending an $ABORT$ message to all the slaves. In the latter case, the master commits the transaction by sending a $COMMIT$ message to all the slaves. Figure 5.2 shows the sequence of messages for a distributed transaction that successfully commits using the two-phase commit protocol.

In our implementation shown in Appendix B, we set up a pseudo-transaction between two processes with one assuming the role of master and the other the role of slave. Each process maintains a variable called $meReady$. Just before the master starts the commit protocol it sets $meReady$ to $true$ and then to $false$ when the commit protocol is finished. When a slave receives a $PREPARE$ message, if it is ready to commit, it set $meReady$ to $true$ and then to $false$ after it has either committed or aborted. Therefore, we can conclude that if a distributed transaction commits successfully, it should always be true that there exists a global state where all the processes involves in the transaction, the master and slaves, are in a global state with $meReady$ set to $true$. With the SCP predicate

$$SCP: \ (meReady == true \,\text{in}\, master) \wedge (meReady == true \,\text{in}\, slave)$$

it is possible to check if such a global state exists for all possible event orderings corresponding to a run. If the SCP predicate is not satisfied, then we know that there is an error in the two-phase commit protocol.

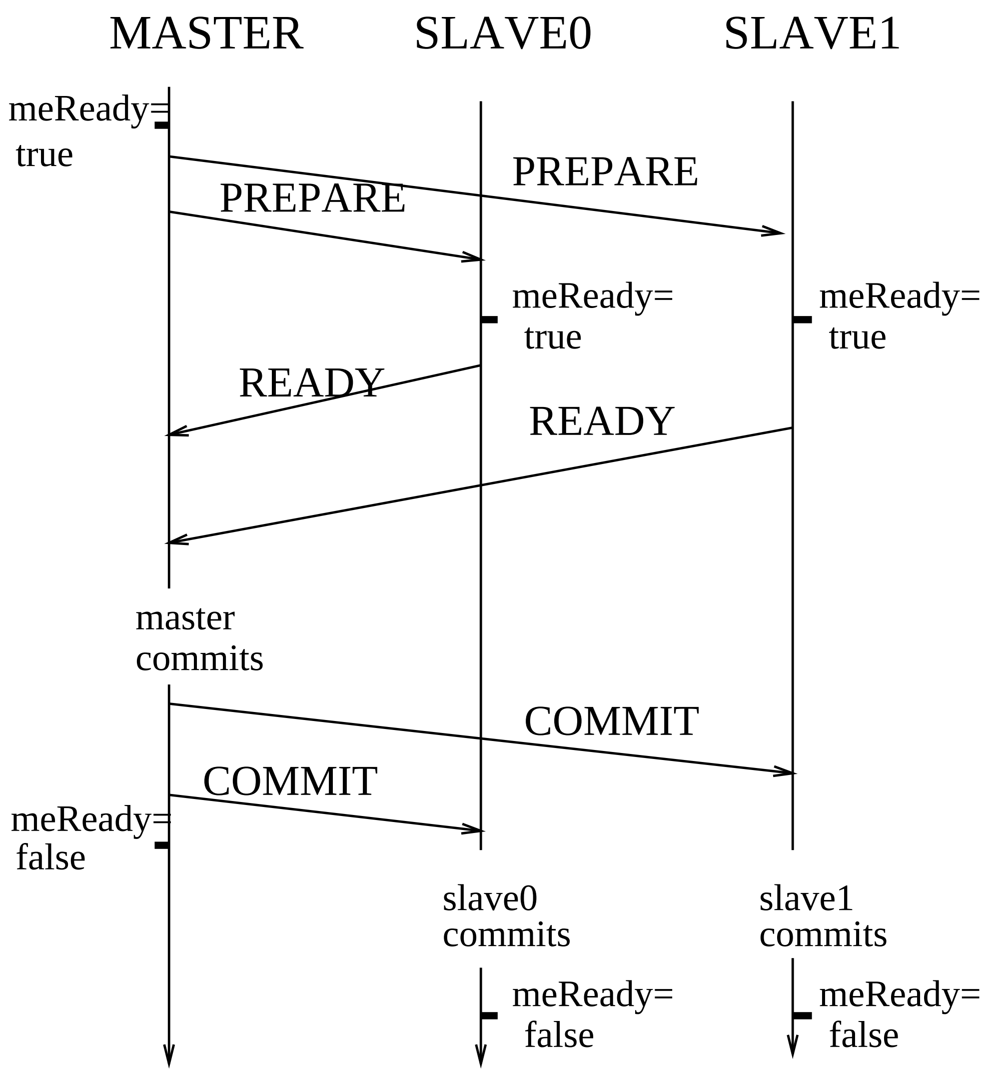

Figure 5.2: Example of two-phase distributed commit protocol

5.3 Distributed Election Program

The goal of the election program is to elect a *leader* from among n processes. Why would one want to elect a leader? Some distributed algorithms are asymmetric and require a leader or coordinator.

Our implementation of the election problem is shown in Appendix C. We are assuming the processes are configured as a ring as shown in Figure 5.3 and that the leader found by our algorithm will be the process with the highest id. Each process maintains a variable called max which represents the highest id the process has seen since the start of the algorithm. The algorithm is spontaneously initiated by one or more processes by sending a $FIND\_HI$ message along with its id to its left neighbor. When a process receives a $FIND\_HI$ message with id msg_hi, it first compares msg_hi to its own id. If they are the same, then that process is the leader. On the other hand, if $msg\_hi > max$ then the process sets $max = msg\_hi$ and sends a $FIND\_HI$ message along with max to its left neighbor else the message is ignored.

When a process discovers it is a leader, it then has to inform the other processes of this. It sends a $FOUND\_HI$ message containing its id to its left neighbor. When a process receives the $FOUND\_HI$ message it makes note of who the leader is and then sends the message to its left neighbor. Referring to Figure 5.3, the order in which the processes find out the identity of the leader should always be: 44, 11, 22, 33. We can verify this with an SLP predicate. In the program we have added the variable $foundit$ to each process. When a process has discovered who the leader is, it toggles $foundit$ to $true$ and then immediately toggles it back to $false$. The following SLP predicate verifies that the processes will always find the leader in the order: 44,11,22,33.

$$SLP: \quad (foundit = true\,\text{in}\,44) \rightarrow (foundit = true\,\text{in}\,11)$$
$$\rightarrow (foundit = true\,\text{in}\,22) \rightarrow (foundit = true\,\text{in}\,33)$$

If the predicate is not detected then we know there is a discrepancy between the expected and actual behavior of the program.

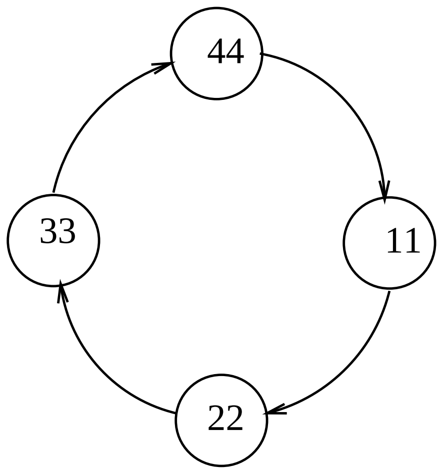

Figure 5.3: Ring configuration used by election program.

Chapter 6

Conclusion and Future Work

This thesis has presented the design and implementation of an online debugger that detects global predicates in distributed programs. Detecting global predicates is a difficult problem because the global state of a distributed program is divided among the processors in a distributed system. The thesis has also presented the design of an environment for creating the event-driven distributed programs that can be debugged by our online debugger.

The debugger detects a subset of the predicates that are defined by the logic presented by Waldecker and Garg in [25]. These predicates are unstable in the sense that it may not be possible to detect them using a distributed snapshot algorithm. Instead, vector time is used to infer the partial ordering of the interesting local states of a distributed program which, in turn, lets us reason about the possible global states that can occur in a run.

Observation and experience has taught us that debuggers for sequential programs are used to detect a particular state of a program and to observe the behavior of the program over time. It is our belief that it would be useful if debuggers for distributed programs could be used in analogous ways since they have proven effective in the past and because the techniques are well-understood and therefore readily accepted by potential users. The logic we use for defining predicates lets us specify distributed predicates that detect global states and, using temporal operators, that examine the behavior of the program over time. In particular, of the four types of predicates detected by this debugger, the *strong conjunctive predicate* (SCP) and *weak conjunctive predicate* (WCP) are used to detect a global state whereas the *strong linked simple predicate* (SLP) and *weak linked simple predicate* (WLP) are used to examine the temporal behavior of a program.

To make the work more complete, we should focus additional research on developing algorithms to detect strong and weak forms of the following global predicates which are defined on a global trace.

$$x + y + z > CONSTANT$$

$$x + y + z < CONSTANT$$

Examples of where the predicates would be useful are when monitoring the allocation of distributed resources, detecting if tokens ever get lost in a distributed algorithm or verifying that a distributed semaphore implementation works correctly[23].

In the future, we want to integrate the offline debugger designed by Chin in [3] with our online debugger. Since we already provide the ability to detect simple predicates, it should be easy to modify the monitors to create trace files that can be used by the offline debugger. We also want to make the message channels less restrictive in terms of message ordering. Currently, the network protocols being used enforce a FIFO ordering, but the predicate detection algorithms presented in chapter 3 will work even for a less-restrictive non-FIFO ordering.

Our monitors utilize gdb to detect simple predicates. We want to take further advantage of gdb and allow the programmer to use it to explicitly singlestep and examine the state of the individual processes when the distributed computation is stopped after a breakpoint.

The way we detect simple predicates is by checking the state of a process after each step which is costly. We would like to modify the way the monitors detect a special subset of the simple predicates called "PC" simple predicates. These are simple predicates of the form,

$$(Program\, Counter \;=\; Address\, X).$$

which correspond to reaching a line or function in a program. Since most architectures provide special hardware support, such as breakpoint traps, to minimize the cost of detecting these special predicates, we want our monitors to take advantage of the hardware support when trying to detect "PC" simple predicates. The following is an example that demonstrates how this type of simple predicate can be used in a global predicate that detects if two processes can enter their critical sections at the same time.

$$\begin{array}{ll} WCP: & (Program\, Counter = critSection()\; in\; Process\, 1) \quad \wedge \\ & (Program\, Counter = critSection()\; in\; Process\, 2) \end{array}$$

Building distributed programs is a difficult and arduous task. As a result, programmers need paradigms that make the job easier even if it means sacrificing efficiency. Two such paradigms are synchronous messages and remote procedure calls (RPC). Synchronous messages prevent a process from doing any work in the interval from when a message is sent until it is finally received. RPCs shift the computation model from one that relies on the explicit I/O primitives of sends and receives to one that uses procedure calls, a computation model that is well-understood and therefore easier to use. Can the techniques we are currently using to debug asynchronous distributed programs be used for distributed programs that use these paradigms? The answer is yes. There is nothing inherent in the logic we use to define predicates or the predicate detection algorithms we use that requires one to use asynchronous messages. In fact, we can model synchronous messages, RPCs and multi-casting using asynchronous messages.

The C source code for the debugger we have designed and built is presented in Appendix **??** and Appendix **??**, and a user's manual is presented in Appendix D.

Appendix A

Distributed Mutual Exlcusion Application

```
/*------------------------------------------------------------*/
/*  DISTRIBUTED MUTUAL EXCLUSION PROGRAM.                     */
/*   This program implements a distributed mutual exclusion */
/*   algorithm.  For the algorithm to work correctly, there */
/*   must exists a logical message channel between every    */
/*   pair of processes.                                     */
/*   A process initiates a request for its critical section */
/*   when you type "t" at the process.                      */
/*                                                          */
/*   For this program, there exists an error at (NOTE 1).   */
/*   The Relation operator should have been 'less than'     */
/*   instead of 'less than or equal'.                       */
/*   This is an error than can be detected with a           */
/*   WCP predicate.                                         */
/*                                                          */
/*------------------------------------------------------------*/

#include "user.h"
#include <sys/time.h>
#include <sys/timeb.h>


#define  REQUEST  30    /* user defined message types */
#define  REPLY    31


extern Proc *peers;
extern  myId;

/*---------------------------------------------------*/
/*   Global user variables                           */
/*---------------------------------------------------*/
int  inCs=0;
int  wantCs=0;
int  reply;
int  CS_clock=0;
```

```
int  request_ts;
int  numProc;
int  numWaiting;
int  wait[MAX_PROCESSES];
int  i;


void  processEvents(msg,event,source)
   char  *msg;
   int    event;
   int    source;
{
Proc  *ptr;
int   *in_request_ts;

switch (event)
   {
                /* Handler routine for REQUEST message */
   case REQUEST:
      in_request_ts=(int *)msg;
      printf("Received REQUEST event");
      if ((*in_request_ts <= request_ts)      /* NOTE 1: ERROR  */
       || (*in_request_ts == request_ts && source<myId)
       || (wantCs==0))
         {
         sendMess(source,REPLY,NULL,NULL);
         }
      else
         {
         wait[numWaiting++]=source;
         }

      if (*in_request_ts>=CS_clock)
         {
         CS_clock = *in_request_ts+1;
         }
      break;

              /* Handler routine for REPLY message */
   case REPLY:
      reply=reply+1;
      printf("Received REPLY event:  reply=%d\n",reply);
      if (reply==numProc)
         {
         printf("**Entering Critical section**\n");
         inCs=1;
```

```
         /*

              CRITICAL   SECTION


         */
         printf("**Leaving Critical section**\n");
         inCs=0;
         wantCs=0;
         for(i=0;i<numWaiting;i++)
            {
            sendMess(wait[i],REPLY,NULL,NULL);
            }
         }
      break;

              /* Handler routine for standard input command */
   case  STD_IN:
      if (msg[0]=='t'  && wantCs==0)
         {
         printf("Received STD_IN event:  request for CS\n");
         wantCs=1;
         request_ts=CS_clock++;
         /* Send REQUEST message to all processes */
         for (ptr=peers,numProc=0;ptr!=NULL;ptr=ptr->next,numProc++)
            {
            sendMess(ptr->iD,REQUEST,&requestTime,sizeof(int));
            }
         reply=0;
         numWaiting=0;
         }
      break;
   }

} /* end processEvents() */
```

Appendix B

Distributed Two-Phase Commit Application

```
/*----------------------------------------------------------*/
/*  DISTRIBUTED TWO-PHASE COMMIT PROGRAM                    */
/*   This program sets up a pseudo-transaction between      */
/*   two processes.  Then, a two-phase commit protocol      */
/*   is utilized to commit the distributed transaction.     */
/*                                                          */
/* Only two processes that are connected by a logical       */
/* message channel can establish a pseudo-transaction.      */
/* To set up a transaction between two processes,           */
/* type "t num" at on of the processes where "num" is       */
/* the logical id of the other process.                     */
/*                                                          */
/* To cause the transaction to commit, type "c" at one      */
/* of the processes.  The process that you enter "c" at     */
/* will assume the role of MASTER and the other process     */
/* will assume the role of SLAVE in the 2-phase commit      */
/* protocol.                                                */
/*                                                          */
/*----------------------------------------------------------*/

#include "user.h"
#include <sys/time.h>
#include <sys/timeb.h>
#define   PREPARE  30
#define   READY    31
#define   COMMIT   32
#define   ABORT    33

#define   NONE     34
#define   TRANSACTION 35

extern Proc  *peers;
extern int    myId;


/*----------------------------------------------------*/
```

```
/*   Global user variables                           */
/*---------------------------------------------------*/
int   meReady;
int   state=NONE;
int   nextTrans=100;
int   transaction;
int   who;
char  date[30];
int   clock;
int   start;
struct timeb  tp;

int   testing;



void  processEvents(msg,event,source)
   char  *msg;
   int    event;
   int    source;
{
int   n;
int   ts[2];

switch (event)
   {

                /* Handler routine for TRANSACTION message */
 case  TRANSACTION:
     {
     transaction = *(int *)msg;
     who=source;
     state=NONE;
     break;
     }
               /* Handler routine for PREPARE message */
 case  PREPARE:
     {
     printf("** Received  PREPARE  message **\n");
     if (*(int *)msg==transaction && state!=ABORT && source==who)
        {
        printf("Ready to commit\n");
        meReady=1;
        state=READY;
        sendMess(who,READY,&transaction,sizeof(int));
        }
```

```
    else
       {
       printf("Transaction %d has been aborted\n",*(int *)msg);
       sendMess(who,ABORT,&transaction,sizeof(int));
       }
    break;
    }
          /* Handler routine for ABORT message */
case ABORT:
   {

   if (transaction==*(int *)msg)
      {
      state=ABORT;
      printf("abort transaction %d\n",transaction);
      }
   meReady=0;
   break;
   }
          /* Handler routine for READY message */
case READY:
   {
   printf("** Received  READY  message ** \n");
   if (transaction==*(int *)msg && state==PREPARE)
      {
      printf("I'm committing transaction %d\n",transaction);
      ts[0]=transaction;
      ts[1]=start;
      sendMess(who,COMMIT,ts,sizeof(int)+sizeof(int));
      ftime(&tp);
      clock = (int) tp.time;
      printf("clock = %d, interval = %d\n",clock,clock-start);
      }
   else
      {
      printf("aborting transaction %d\n",*(int *)msg);
      sendMess(source,ABORT,msg,sizeof(int));
      }
   meReady=0;
   break;
   }
              /* Handler routine for COMMIT message */
   case COMMIT:
      {
      printf("** Received  COMMIT  message ** \n");
      if (transaction==*(int *)msg && (state==READY))
```

```
        {
        printf("I'm committing transaction %d\n",transaction);
        state==COMMIT;
        msg=msg+sizeof(int);
        start = *(int *) msg;
        ftime(&tp);
        clock = (int) tp.time;
        printf("clock = %d, interval = %d\n",clock,clock-start);
        }
     else
        {
        printf("I'm aborting transaction %d\n",transaction);
        state==ABORT;
        }
     meReady=0;
     break;
     }
             /* Handler routine for standard input command */
case  STD_IN:
   if (msg[0]=='t')
      {
      transaction=nextTrans++;
      state=NONE;
      who = atoi(msg+1);
      printf("start transaction %d with %d\n",transaction,who);
      sendMess(who,TRANSACTION,&transaction,sizeof(int));
      }
   else if (msg[0]=='a')
      {
      state=ABORT;
      }
   else if (msg[0]=='c')
      {
      if (state==NONE)
         {
         ftime(&tp);
         start = (int) tp.time;
         printf("start time=%d\n",start);
         meReady=1;
         printf("preparing to commit\n");
         state=PREPARE;
         sendMess(who,PREPARE,&transaction,sizeof(int));
         }
      }
   else if (msg[0]=='p')
      {
```

```
        printf("transaction = %d, who = %d\n",transaction,who);
        }
     break;

     }
}
```

Appendix C

Distributed Election Application

```
/*----------------------------------------------------------*/
/*  DISTRIBUTED ELECTION PROGRAM on a ring                  */
/*    This program finds a leader from among a set of       */
/*    of processes configured as a ring.                    */
/*    It will choose the process with the highest logical  */
/*    id as the leader                                      */
/*                                                          */
/*  In the ring, all messages are passed to the left        */
/*  neighbor.  The global variable, sendId, stores the      */
/*  logical id of left neighbor. Note, there must exist     */
/*  a logical channel between the process and its left      */
/*  neighbor. To assign a process a left neighbor,          */
/*  type "l  left_neighbor_id".                             */
/*  Once all the nodes are hooked into a ring, type         */
/*  "k" at one or more of the processes to initiate         */
/*  the algorithm.                                          */
/*                                                          */
/*----------------------------------------------------------*/

#include "user.h"
#define  FIND_HI   30      /* user defined  message  types */
#define  FOUND_HI  31

extern   Proc   *peers;
extern   int     myId;
/*----------------------------------------------------*/
/*   Global user variables                            */
/*----------------------------------------------------*/
int    sendId;
int    max;
int    start = NULL;
int    foundit;
/*----------------------------------------------------*/
/*  prototype functions                               */
/*----------------------------------------------------*/
void  findHi();
```

```
void  foundHi();



void  processEvents(msg,event,source)
   char  *msg;
   int    event;
   int    source;
{
int   n;

switch (event)
   {
               /* Handler routine for FIND_HI message */
   case FIND_HI:
      findHi(msg);
      break;
               /* Handler routine for FOUND_HI message */
   case FOUND_HI:
      foundHi(msg);
      break;
              /* Handler routine for standard input command */
   case STD_IN:
      if (msg[0] == 'k')   /* start election algorithm */
         {
         printf("\n starting findHi algorithm \n");
         n=-1;
         findHi(&n);
         }
      else if (msg[0] == 'l') /* assign left neighbor */
         {
         sendId = (atoi(msg+1));
         if (sendId==0)
            {
            fprintf(stderr,"Warning: can't get left neighbor\n");
            }
         else
            {
            printf("Left neighbor is %d\n",sendId);
            }
         }
      break;
   }
} /* end process Events */
```

```
/*-----------------------------------------------------------*/
void  findHi(num)
   int   *num;
{

printf("\n FIND_HI event:  num = %d\n",*num);
if (!start)
   {
   start = 1;
   if (*num > myId)
      {
      max = *num;
      }
   else
      {
      max = myId;
      }
   sendMess(sendId,FIND_HI,&max,sizeof(int));
   }
else
   {
   if (*num > max)
      {
      max = *num;
      sendMess(sendId,FIND_HI,&max,sizeof(int));
      }
   else if (*num == myId)
      {
      foundit=1;     /* toggle foundit true  */
      printf("\n I found it!!! myId = %d\n", myId);
      foundit=0;    /*  then toggle it false */
      sendMess(sendId,FOUND_HI,&myId,sizeof(int));
      }
   }
}
/*-------------------------------------------------------*/
void  foundHi(num)
  int  *num;
{

printf("\n FOUND_HI event:  myId=%d  max id=%d\n",myId,*num);
start = NULL;
if (*num!=myId)
   {
   foundit=1;   /* toggle foundit true  */
```

```
   printf("** %d has found Hi ID = %d\n",myId,*num);
   foundit=0;   /* then toggle it false */
   sendMess(sendId,FOUND_HI,num, sizeof(int));
   }
} /* end foundHi() */
/*-----------------------------------------------------------*/
```

Appendix D

Using The Online Debugger

This manual will briefly describe how to use the debugger. It will present the following topics:

- How to start and initialize the debugger and the distributed computation.
- How to setup debugger to detect distributed predicates.

D.1 Starting the Debugger and Distributed Program

When running the debugger, we would advise keeping the following programs and files in the same directory: "coordinator", "monitor", "application" and the application's source files. Note, the application must be compiled in debug mode.

The first step to starting the debugger is to start the coordinator. In a window, type the following at a shell prompt.

```
coordinator
```

The coordinator will then set up a TCP listening port at port number 2018 and wait for the monitor processes to be started. Note, if port number 2018 is already in use the coordinator will exit.

Next, the monitor processes should be started. When a monitor is started, it will automatically start two additional processes, the gdb debugger and an application process. To start a monitor, type the following command at a shell prompt.

```
monitor logid -hostname
```

The input parameter `logid` is the logical id assigned to the application process and `hostname` is the name of the host where the coordinator is running. Note, the logical id assigned to each application process in a distributed computation should be unique.

The following is a list of the predefined commands accepted by the coordinator.

.l s d	setup bidirectional channel or link between the application processes with id s and d
.p	ping all the application processes
.s	check status of all the application processes
.r	reset vector clock for all the application processes
.a	adjust length of vector clock for all application processes
.m	put all monitors into *record mode.* This also indicates that the distributed computation should be rolled back if a WCP predicate is detected
.d	place coordinator into "distributed predicate input mode."

The following is a list of predefined commands accepted by an application process

.p	ping the coordinator

.s	get a list of all neighbors processes the application is directly linked to
.q	ping all neighbors

D.2 To Detect a WCP Predicate

If you want the distributed computation to halt in a global state where the WCP occurred (by rolling back), input the following commands at the coordinator in the order listed once all the monitors have been started.

1. Type `.m`, to put system into record mode
2. Type `.a`, to adjust length of vector clocks
3. Type `.d`, then input the WCP predicate to detect.
4. Type `.r`, to reset vector clocks and to start the monitoring application
5. Explicitly establish links between process that should be able to directly communicate using the *link* command described above.

At this point, the coordinator will inform you when the $WCP(prod)$ predicate is detected and it will also rollback the distributed computation to a consistent global state where *prod* is *true*. If you do not want the computation to rollback, then skip item 1 above.

D.3 To Detect a SCP Predicate

To detect a SCP predicate, input the following commands at the coordinator in the order listed once the monitors have been started.

1. Type `.a`, to adjust the length of the vector clocks

2. Type `.d`, then input the SCP predicate to detect

3. Type `.r`, to reset vector clocks and to start monitoring application

4. Establish links between application processes with the *link* command.

It is important that the first three be done in the order listed. The last item can be done at any time.

D.4 To Detect a SLP Predicate

To detect a SLP predicate, input the following commands at the coordinator in the order listed once the monitors have been started.

1. Type `.a`, to adjust the length of the vector clocks

2. Type `.d`, then input the SLP predicate to detect

3. Type `.r`, to reset vector clocks and to start monitoring application

4. Establish links between application processes with the *link* command.

It is important that the first three be done in the order listed. The last item can be done at any time.

BIBLIOGRAPHY

[1] Peter C. Bates. "Effective Instrumentation is the Key to Effective Monitoring". In *Proceedings of the International Conference on Distributed Computing Systems*, 1991.

[2] K.M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM TOCS*, 3(1):63–75, February 1985.

[3] Bryan Chin. "Offline Debugging of Distributed Processes". Master's thesis, University of Texas at Austin, December 1991.

[4] R. Cooper and K. Marzullo. "Consistent Detection of Global Predicates". In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

[5] Michael L. Dertouzos. "Building the Information Marketplace". *Technology Review*, pages 29–40, January 1991.

[6] E.A. Emerson, E.M. Clarke, and A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, pages 244–263, 1986.

[7] J. Fowler and W. Zwaenepoel. "Causal Distributed Breakpoints". In *Proceedings of the International Conference on Distributed Computing Sys-*

tems, pages 134–141. IEEE, 1990.

[8] Dieter Haban and Wolfgang Weigel. "Global Events and Global Breakpoints in Distributed Systems". In *The Hawaiian International Conference on System Sciences*, pages 166–175, 1988.

[9] S. Haldar and K. Vidyasankar. "Counterexamples to a One Writer Multireader Atomic Variable Construction of Burns and Peterson". *Operating Systems Review*, 26(1):78–88, 1992.

[10] W. Hseush and G. Kaiser. "Modelling Concurrency in Parallel Debugging". *ACM Sigplan Notices*, 25(3):11–20, March 1990.

[11] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". *Communications of the ACM*, pages 558–565, July 1978.

[12] William S. Lloyd and Phil Kearns. "Global State Considered Harmful: Testing Causal Relationships in Distributed Programs". In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 245–247, 1991.

[13] F. Mattern. "Virtual Time and Global States of Distributed Systems". In *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publshers, 1989.

[14] Charles E. McDowell and David P. Helmbold. "Debugging Concurrent Programs". *ACM Computing Surveys*, 21(4):593–622, December 1989.

[15] Barton P. Miller and Jong-Deok Choi. "Breakpoints and Halting in Distributed Programs". In *Proceedings of the International Conference on Distributed Computing Systems*, pages 316–323. IEEE, 1988.

[16] Barton P. Miller and Charles McDowell. "Summary of ACM/ONR Workshop on Parallel and Distributed Debugging". *Operating Systems Review*, 26(1):18–31, 1992.

[17] D.Z. Pan and M.A. Linton. "Recap". In *Proceedings of the ACM SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, 1988.

[18] G. Ricart and A.K. Agrawala. "An Optimal Algorithm for Mutual Exclusion in Computer Networks". *Communications of the ACM*, 24(1):9–17, January 1981.

[19] A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, MA, 1991.

[20] M. Spezialetti. "An Approach to Reducing Delays in Recognizing Distributed Event Occurances". In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 151–162. ACM/ONR, 1991.

[21] M. Spezialetti and J.P. Kearns. "Simultaneous Regions: A Framework for the Consistent Monitoring of Distributed Systems". In *Proceedings of the International Conference on Distributed Computing Systems*, pages 61–69. IEEE, 1989.

[22] Zhigong Thong. "Crash Recovery of Distributed Systems", 1992. Master's Report, University of Texas at Austin, under preparation.

[23] A.I. Tomlinson. "Lecture on an Algorithm for Maintaining Global Predicates", March 1992.

[24] Brian Waldecker. *"Detection of Unstable Predicates in Debugging Distributed Programs"*. PhD thesis, University of Texas at Austin, May 1991.

[25] Brian Waldecker and Vijay K. Garg. "Detection of Strong Unstable Predicates in Distributed Programs". Technical report, University of Texas at Austin, 1991.

[26] D.A. Young. *X Window Systems Programming and Applications with Xt.* Prentice Hall, New Jersey, 1989.

VITA

Greg M. Hoagland was born in Endicott, New York in 1967. He attended Georgetown High School in Georgetown, Texas and received a Bachelor of Science in Electrical Engineering from the University of Texas at Austin in August 1990. Greg can be contacted at Qualcomm Inc., 10555 Sorrento Valley Road, San Diego, California 92121.

Permanent address: 907 Saunders Drive
Round Rock, Texas 78664

This thesis was typeset [1]with LaTeX by author.

[1]The LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin thesis format was written by Khe-Sing The.