Debugging in a Distributed World: Observation and Control

Ashis Tarafdar *
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188, USA
ashis@cs.utexas.edu

Vijay K. Garg †
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

Abstract

Debugging distributed programs is considerably more difficult than debugging sequential programs. We address issues in debugging distributed programs and provide a general framework for observing and controlling a distributed computation and its applications to distributed debugging. Observing distributed computations involves solving the predicate detection problem. We present the main ideas involved in developing efficient algorithms for predicate detection. Controlling distributed computations involves solving the predicate control problem. Predicate control may be used to restrict the behavior of the distributed program to suspicious executions. We also present an example of how predicate detection and predicate control can be used in practice to facilitate distributed debugging.

1. Introduction

While developing software, we sometimes experience a *software failure* – a mismatch between expected and actual software executions. *Debugging* is the process of tracking down the source of such a software failure. While the skill and intuition of the software developer play a major part in the debugging process, an indispensable factor is the effective use of software tools that provide an environment for observing and controlling executions. Such tools, known as *debuggers*, have been widely used for traditional sequential software development. However, the trend towards distributed software leads to issues which make traditional debuggers inadequate. It is to identify and address these issues that the field of distributed debugging has received wide attention.

*supported in part by the MCD Fellowship
†supported in part by the NSF Grants ECS-9414780, CCR-9520540, TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

Distributed debugging is an ongoing research area. The increasing number of distributed programs together with the inherent difficulty in writing such programs and the limited support currently provided for distributed debugging would lead to a potentially catastrophic increase in the number of unreliable distributed programs. Therefore, there is a constant search for new and better techniques for distributed debugging.

We divide the actions of a debugger into two categories – *observation* and *control*. These are exemplified by the most common operations in traditional debuggers: setting breakpoints (control) and observing variable values (observation). In distributed systems, observation and control are made more difficult by the distributed nature of the executions involved.

Our goal is provide a general framework for observing and controlling a distributed computation and to demonstrate the effectiveness of this framework in distributed debugging. This paper is not a complete survey of the distributed debugging field; rather it is a summary of those aspects of distributed debugging that are part of the focus of the work done at the Parallel and Distributed Systems Laboratory (PDSLAB) at the University of Texas at Austin.

The paper is organized as follows. Section 2 presents a summary of current distributed debugging techniques in practice and in research. Section 3 describes our model of distributed computation and global predicates. Section 4 discusses key problems and their solutions for observation of global properties. Section 5 discusses various models of control and reports some of our work for control. Finally, Section 6 describes an example of debugging a distributed system.

2. Brief Survey

While debuggers in use today (e.g. gdb, dbx, Visual C++'s debugger) provide limited support for multithreaded concurrency, they cannot support distributed programs which run in multiple address spaces. A limited form

of monitoring may be achieved by running multiple debuggers on each of the processes. However, even so, basic debugging features such as deterministic replay and global breakpoints are not supported.

In spite of the lack of support for distributed debugging in commercial tools, there have been many research projects that have built either prototype or full-fledged implementations. An extensive summary of these projects may be found in [13]. Closely related to debugging is the field of testing which studies the process of selecting test cases. A good summary of testing techniques for distributed programs may be found in [15].

Aside from practical implementation work, much research has been devoted to the study of problems arising in distributed debugging. As noted before, we classify these problems into those that deal with observation and those that deal with control.

Predicate Detection [1, 6] is the main problem involved in observing distributed computations. It involves detecting whether a specified global property ever occurs in a distributed computation. For example, one might wish to check that the validity of mutually exclusive sections of code is maintained. Predicate detection is used to set global breakpoints in distributed computations. Approaches to solving predicate detection are divided into three categories: global snapshot based [2], lattice construction based [4], and predicate restriction based [7] approaches. The first approach can detect only *stable* predicates (which remain true once they become true). The second approach uses the interleaving model of concurrency and, therefore, suffers from combinatorial explosion. The last approach uses a partial order model and limits itself to classes of predicates which can be detected efficiently. Our focus in this paper will be on the third approach to predicate detection.

Control of distributed computations can be at various levels. So far, the literature has focussed mainly on the most basic form of control: deterministic replaying of distributed computations to recreate failures [10, 14]. This leads to a debugging cycle consisting of passive observation and computation replaying. We believe that a more effective and active debugging method would involve instead a cycle of observation followed by controlled replaying based on observation [17]. We may classify control into *on-line control* or *off-line control* depending on whether it is applied to a fresh computation or to a replayed computation. Off-line control is easier to achieve than on-line control because of the pre-knowledge of future events. One method to control a computation is to re-order messages. This form of control (which may also be considered a testing methodology) has been studied in both its on-line [12] and off-line [8] variants. We consider a less-intrusive form of control which is only capable of delaying events (and not reordering them). This has been termed the *predicate control* problem and has been studied in both on-line and off-line variants [17].

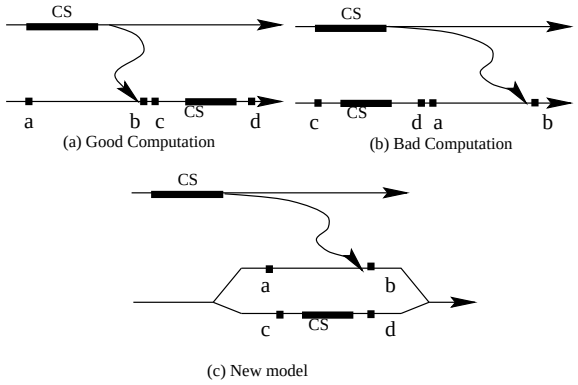

Figure 1. Addressing False Causality

3. Modeling Distributed Computations

The traditional model of a distributed computation has been based on the *happened before* relation introduced in [9]. The distributed system is divided into a set of processes. Events on a single process are totally ordered according to the local clock from the initial (earliest) event to the final (latest) event. Processes are assumed to communicate only through the use of messages and don't share a common clock or common state. Therefore, events across processes can only be partially ordered based on whether an event can possibly "know" about another through a chain of messages. To summarize, an event e_1 happened before another event e_2 if and only if e_2 follows e_1 on the same process or e_2 can "know" about e_1 through some chain of message communications. We also say that two events are *concurrent* if neither happened before the other. Since, an event can cause another event *only if* it happened before the other event, the happened before relation is said to model *potential causality* between events.

The happened-before model of a computation is easy to implement while tracing a running distributed computation. This involves the use of vector clocks [11]. A vector clock is stored for each event. By comparing the vector clocks of two events, we can deduce their ordering with respect to the happened-before relation.

Although the happened-before model has been the basis for most of the work in distributed debugging, it has often been criticized for allowing false causality. This is illustrated in the example in Figure 1. Consider running a distributed mutual-exclusion program. The happened-before model of the resulting distributed computation is shown in Figure 1(a). If mutual-exclusion violation is the predicate that we are trying to detect, it would not be detected because the message ensures that the two critical sections cannot occur at the same time. However, in actuality, the mes-

sage may have been fortuitous and the sections of execution marked by intervals (a, b) and (c, d) may have been independent (for example, independent threads). The scenario in Figure 1(a) may be just one possible scheduling of events. Figure 1(b) shows another scheduling in which mutual-exclusion would be violated.

A model which partially orders the events on a local process would allow events within a process to be independent. Figure 1(c) shows such a model for the example. This representation models both of the previous schedulings. In general, there would be an exponential number of happened-before representations corresponding to a single representation in the new model. We call the new model a *strong causality diagram* [16].

Strong causality is based on the *strong causally precedes* relation between events. It is an extension of happened before based on the key observation that totally ordering the events on a single process leads to false causality between events. Therefore, it allows events on a single process to be partially ordered. This allows for better modeling of independence between local events, which we have just seen is a key factor in detecting more bugs. This is especially important in view of the prevalence of multi-threaded programs, in which events in the same process but on different threads are often independent.

The strong causally precedes relation is also easy to implement while tracing a running distributed computation. It involves a generalization of vector clocks that was introduced in [5] which allow the tracing of a general partial order. Comparing the clocks of two events gives us their relative ordering according to the strong causally precedes relation.

The choice of how much independence to model depends on the particular application. As we will see, problems become harder to solve in the new strong causality model but the model itself is more expressive. This expressiveness is demonstrated in both observation and control. During observation, as illustrated in the mutual exclusion example, strong causality allows us to detect more bugs [16]. During control, strong causality may be used to model independent events and thus model various message reorderings in the same model. A form of this kind of modeling may be found in [8]. For our purposes of controlling using event delays, the happened before model suffices.

Since each event leads to a state, we may apply both happened before and strong causally precedes to order states instead of events. In such a case, we define a *global state* to be a complete set of local component states and a *consistent global state* to be a global state in which all local states are mutually unordered with respect to the relation under consideration.

A *global predicate* is a boolean function on the set of global states. Such predicates will be used in the predicate detection and predicate control problems.

4. Observing Computations: Predicate Detection

Predicate detection involves observing a computation to detect the existence of a global state that satisfies a given predicate. Checking every global state by explicit construction would lead to combinatorial explosion and so we follow the approach of trying to avoid this by working in the partial order model rather than attempting to construct all possible interleavings (an approach that has been studied [4]).

4.1. NP-Completeness and Conjunctive Predicates

Even within the happened before model, if the predicate is a boolean expression it has been demonstrated that the predicate detection problem is NP-complete [3]. In spite of this fact, there are efficient algorithms for predicate detection for several useful classes of predicates. For a survey of such algorithms, refer to [6].

In particular, the class of *conjunctive predicates* is found to be very useful in practice. These are predicates that can be expressed as a conjunction of local predicates. Intuitively, these properties check if a combination of events happen together. An example of a global predicate would be detecting if functions on two processes are entered at the same time, violating a required mutual-exclusion property.

4.2. Detecting Conjunctive Predicates in the Happened Before Model

The detection of conjunctive predicates has been efficiently solved in [7]. Their algorithm first eliminates all states in which the local predicate is false. It then starts at the initial global state and proceeds forward. It successively considers global states in such a manner that it is not necessary to consider all interleavings and the total number of global states considered is linear in the size of the happened before model. The key observation that allows this is that if, in a global state under consideration, two states s and t are related by happened before (s happened before t), then s can never be in a consistent global state because it would also happen before any local state following t on the same process. So s may be eliminated and a new global state may be constructed. At every step we eliminate some local state from consideration which ensures a linear number of steps. The whole algorithm requires $O(n^2m)$ comparisons, where n is the number of processes and m is a bound on the number of send and receive events in a process.

4.3. Detecting Conjunctive Predicates in the Strong Causality Model

As noted before, the strong causality model allows us to express independences between events on a local process, thus leading to better bug detection. In the strong causality model, we cannot apply the same algorithm for conjunctive predicate detection because the key observation which allowed us to eliminate a local state from consideration in every global state considered would not work as before. In fact, the problem becomes NP-complete as is demonstrated in [16].

However, for a large class of computations, we can still efficiently detect conjunctive predicates. If, in a computation, the events which send messages are all totally ordered with respect to each other, then we say that the computation is *send-ordered*. Analogously, we may define *receive-ordered* computations as those which have totally ordered receive events. Now for the classes of send-ordered and receive-ordered computations, we can efficiently detect conjunctive predicates. The algorithm to do so is presented in [16]. It applies a preprocessing step to the strong causality model to convert it to a representative happened before model on which the above algorithm for conjunctive predicates may be applied.

5. Controlling Computations: Predicate Control

The form of control that we impose on the distributed computation is *predicate control* – the maintenance of a global safety property. The debugger has control over the relative execution speeds of the processes but may not reorder the messages. Thus the computation remains the same in terms of the sequence of events on each process. However, among all the possible global states that could possibly occur in the computation, the controlled execution speeds can ensure that certain global states cannot occur.

In terms of the happened before model of a distributed computation, predicate control involves adding extra causality edges to constrain the partial-order to a stricter partial-order. An extra causality edge would enforce that an event on one process must wait for an event on another process. Further, the added causality must not cause cycles with the existing causality.

Following our definition of a consistent global state, the added causal edges would reduce the number of consistent global states. The goal is to add causality edges so that only those global states that satisfy the specified global predicate are consistent. It is also desirable to add as few causality edges as possible since they constrain concurrency in the computation.

An important distinction may be made depending on whether the distributed computation is being controlled for the first run or during a replay. If it is being controlled for the first run, then the partial-order of events is not known *a priori*. Since the partial-order of events is provided on-line, we call this *on-line predicate control*. During a replay, since the partial-order of events is provided off-line in the trace, the control is called *off-line predicate control*. Clearly, on-line predicate control is harder than off-line predicate control.

5.1. NP-Completeness and Disjunctive Predicates

If the specified global predicate to be maintained is a simple boolean expression of local predicates then it can be shown that the problem of determining if a control strategy exists is NP-complete [17]. Considering that the predicate detection problem is also NP-complete for general boolean expressions, this fact is not surprising. The issue of obvious interest is whether there is a useful class of predicates for which predicates can be controlled efficiently.

One such class of predicates is *disjunctive predicates*. These are predicates which may be expressed as a disjunction of local predicates. Intuitively, these predicates state that at least one property must be satisfied at all times or, in other words, that a bad combination of events does not occur. Some examples of useful disjunctive predicates are:

(1) Two process mutual exclusion:
$\neg cs_1 \vee \neg cs_2$
(2) At least one server is available:
$avail_1 \vee avail_2 \vee \ldots avail_n$
(3) x must happen before y:
$after\ x \vee before\ y$
(4) At least one philosopher is thinking:
$think_1 \vee think_2 \vee \ldots think_n$
(5) $(n-1)$-mutual exclusion:
$\neg cs_1 \vee \neg cs_2 \vee \ldots \neg cs_n$

Note how we can even achieve the fine-grained control necessary to cause a specific event to happen before another as in property (3). This was done using local predicates to check if the event has happened yet.

5.2. Solving Off-line Disjunctive Predicate Control

A disjunctive predicate divides the sequence of local states in each process into true and false intervals depending on whether the local disjunct is satisfied. We must ensure that any consistent global state intersects at least one true interval. Clearly, the initial global state must intersect a true interval. Before this true interval can become false, we must wait for another true interval. This translates to adding

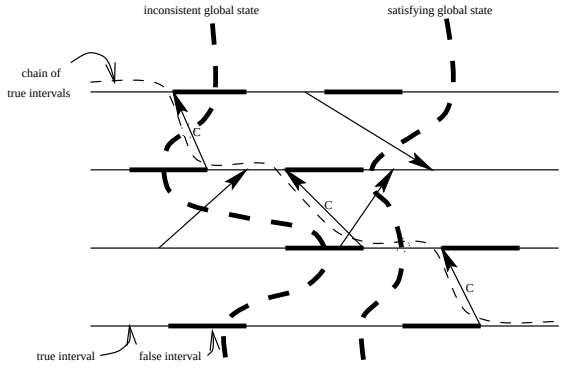

Figure 2. Chain of true intervals

a causality edge from the beginning of the second true interval to the end of the first true interval. When the second true interval is to be exited, we must wait for a third, and so on. This leads to a chain of alternating true intervals and backward pointing added causality edges (Figure 2). Finally, this chain must end in a true interval that includes the last state of some process. Since any global state has to intersect this chain, it must either satisfy the disjunctive predicate or it must be inconsistent.

The only problem that remains is whether the chain of true intervals can be selected efficiently. This is demonstrated by an algorithm in [17] which achieves predicate control if at all possible. The algorithm runs in $O(n^2p)$ where p is a bound on the number of false intervals in a process.

5.3. Solving On-line Disjunctive Predicate Control

Without prior knowledge of the entire partial-order of events, it is impossible to prevent a deadlock while adding causal edges on-line [17]. An adversary can force the debugger into making the wrong choice when presented with multiple possible causal edges to add. Since the debugger is unaware of the future computation, it is unable to avoid creating 'wait-for' cycles with the as yet unrevealed partial-order.

Therefore, we must assume that no process can wait in a false interval. For example, in a two-process mutual exclusion this would mean that a process cannot block in its critical section. Note that this is an assumption that has been made in traditional solutions to mutual exclusion.

Under this assumption, we follow the same approach of constructing a chain of true intervals and backward pointing causality edges. One process starts in a true interval and before it exits the interval, it must wait for another process to enter its true interval. The selection of this process may be made at random or by broadcasting a request to all processes. Since any process must eventually enter a true interval (by the assumption), there will be no deadlocks.

6. Example: Active Debugging of a Replicated Server System

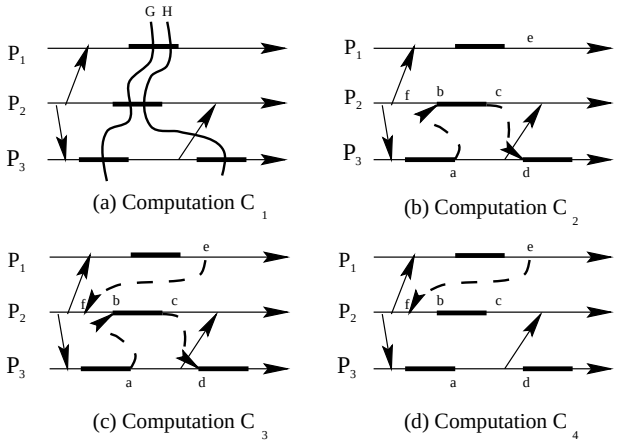

Figure 3. Example: Distributed Debugging

We have provided the ability to observe and control a distributed computation both while being replayed and while being run for the first time. However, the utility of these abilities depends on how effectively they can be used in the debugging process. We now discuss applications of predicate detection, and both off-line and on-line predicate control while debugging distributed programs.

Our running example will be a replicated server system with three server processes P_1, P_2 and P_3. During debugging, a trace of the distributed computation C_1 was taken as shown in Figure 3(a). The thicker intervals in the process executions indicate intervals when the servers weren't available for service.

The system should have been designed to ensure that one server was available at all times. So we run a predicate detection algorithm on C_1 (such as that in Section 4.2) to detect bug_1: "all the servers are unavailable". We detect two consistent global states G and H, as shown in the diagram, where bug_1 is possible.

Our next step is to control C_1 with the safety predicate:"at least one server must be available at all times". Since this is a disjunctive predicate, we may use our off-line algorithm to control C_1, and the resulting computation C_2 is shown in Figure 3(b). Note how the control messages from a to b and from c to d ensure that global states G and H are no longer consistent and bug_1 doesn't occur.

We now suspect bug_2: "states f and e occur at the same time". We run the predicate detection algorithm in Section 4.2 to detect that bug_2 is indeed possible in C_2. We now impose the required safety predicate that "e must happen

before f" and control C_2 using our off-line algorithm. The resulting computation C_3 in Figure 3(c) is found to be satisfactory.

However, we suspect that bug_2 may have caused bug_1. We, therefore, return to our first computation C_1 and apply off-line control to it with the safety predicate: "e must happen before f" This leads to computation C_4 in Figure 3(d). Note how the new control message from e to f ensures that G and H are inconsistent. So eliminating bug_2 also eliminates bug_1 and we conclude that bug_2 is the most important bug.

Now that we have discovered a possible bug in the system, we should check all future computations under the constraint that this bug does not occur. We, therefore, apply our on-line algorithm with the predicate: "e must happen before f" while running the system to generate new computations. If no more bugs are detected, our confidence that bug_2 is the problem increases.

In this illustration, we have demonstrated three areas where predicate control may be applied:

- Determining if a bug recurs under added safety constraints. (off-line)

- Determining the most important bug. (off-line)

- Preventing possible bugs in computations being run for the first time. (on-line)

7. Conclusions

Though the focus of this paper has been on distributed debugging, observation and control of distributed computations are useful abstractions that may be used for other problems in distributed systems that are based on modeling distributed computations (e.g. recovery). In this paper, we have presented the difficulties and some solutions for observation and control. We have demonstrated applications of this framework to distributed debugging. By presenting these results, we demonstrate the need for attention to two aspects of distributed debugging that have been largely ignored in the past:
(1) modeling a computation to eliminate false causality and studying how best to apply this model to distributed debugging, and
(2) how to control running computations while debugging, especially while replaying a known computation.

References

[1] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, chapter 4. Addison-Wesley, 1993.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[3] C. Chase and V. K. Garg. On techniques and their limitations for the global predicate detection problem. In *Proc. of the Workshop on Distributed Algorithms*, pages 303 – 317, France, Sept. 1995.

[4] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163 – 173, Santa Cruz, California, 1991.

[5] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28 – 33, August 1991.

[6] V. K. Garg. Observation of global properties in distributed systems. In *Proceedings of the IEEE International Conference on Software and Knowledge Engineering*, pages 418 – 425, Lake Tahoe, Nevada, June 1996.

[7] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[8] R. Kilgore and C. Chase. Re-execution of distributed programs to detect bugs hidden by racing messages. In *Proceedings of the International Conference on System Sciences*, Hawaii, January 1997.

[9] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[10] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[11] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V. (North Holland), 1989.

[12] V. V. Murty and V. K. Garg. Characterization of message ordering specifications and protocols. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, May 1997.

[13] C. M. Pancake and R. H. B. Netzer. A bibliography of parallel debuggers. 26(1):21 – 37, 1991. updated version at http://www.engr.orst.edu/ pancake/papers/biblio.html.

[14] M. Raynal. *Algorithms for mutual exclusion*. MIT Press, 1986.

[15] K. C. Tai and R. H. Carver. *Testing of distributed programs*, chapter 33, pages 955–978. McGraw-Hill. (A. Zoyama Ed.), 1996.

[16] A. Tarafdar and V. K. Garg. Addressing false causality while detecting predicates in distributed programs. In *Proceedings of the 18th International Conference on Distributed Computing Systems (to appear)*, May 1998. available at http://maple.ece.utexas.edu as technical report TR-PDS-1997-011.

[17] A. Tarafdar and V. K. Garg. Predicate control for active debugging of distributed programs. In *Proceedings of the 9th Symposium on Parallel and Distributed Processing (to appear)*. IEEE, April 1998. available at http://maple.ece.utexas.edu as technical report TR-PDS-1998-002.