

Distributed Systems Debugging – State of the Art

Jiří Patera

Informatics and Computer Engineering, 2nd year, internal form

Supervised by Prof. Ing. Jiří Šafařík, CSc.

Department of Computer Science and Engineering

University of West Bohemia in Pilsen, Univerzitní 8, 306 14, Plzeň, Czech Republic.

jipatera@kiv.zcu.cz

Abstract – *Software engineers have to face many problems when creating, testing, and debugging their applications. Even a small modification of a distributed system can considerably change its behavior. Today’s programs in distributed and embedded systems are often designed for long-time running applications and, thus, very complex. It is unlikely that all mistakes in such applications are eliminated during the developing phase. Therefore, such programs are often monitored during the run-time in order to identify and eliminate all previously hidden mistakes. Afterwards, if the erroneous situation has been identified, the execution of the application can be simulated using previously recorded information and the faulty process can be inspected in more detail. This paper investigates different approaches to monitoring, controlling and debugging distributed programs.*

Keywords: *Debugging, testing, monitoring, replay, distributed system.*

1 Introduction

The importance of using distributed systems has been recently increasing not only with the growth of the Internet, but also by their usage in real-time embedded systems. However, today’s complex distributed systems are difficult to design without bugs (mismatches between expected and actual computations). Although, programmer’s skills and intuition play a very important role during the design process, the existence of tools (debuggers), that provide observation and control of a computation, is essential.

Algorithms for observation are often used to stop a distributed program under certain, usually undesirable, conditions. Algorithms for control are then used to restrict the behavior of a distributed program to suspicious executions. In addition, they are also useful when a programmer wants to test the distributed program under simulated conditions.

There are three fundamental characteristics of distributed systems: the lack of global clock, the lack of global state, and the presence of multiple processes.

The outline of this paper is as follows. In Section 2, the general model of a distributed program is investigated. It is followed by the studying of predicates in Section 3. Next, some approaches to observing and controlling a distributed computation are given out in Section 4, and Section 5. Finally, replay of a distributed execution is explored in Section 6.

2 Model of the Computation

A distributed system consists of n sequential processes p_1, \dots, p_n which can send messages to each other over message channels. As we know, each process consists of a series of events.

Thus, we can describe (like in [4]) an execution of each process by its *history*. First k events in a process's history are called a *prefix* of the process's history. We can also form a *global history* of a distributed system as the union of individual process histories.

A *cut* of a system's execution is a subset of its global history which is formed as a union of prefixes of individual process histories.

The cut is *consistent* if for each event it contains, it also contains all the events that happened-before that event. Otherwise, the cut is said to be *inconsistent*. A *consistent global state* is then such a global state that corresponds to a consistent cut.

An *execution* of a distributed system is a series of transitions between global states of the system. In fact, we can have as many global states as the number of events in the system is, some of these states are consistent, and some are not. A *run* is a total ordering in a global history of the system which is consistent with each local history's ordering. Apart from that, a *linearization* (also known as a *consistent run*) is consistent with the global history's ordering. Thus, not all runs pass through consistent global states, but all linearizations do pass through them only.

3 Global State Predicates

Detecting a condition in a distributed system equals to evaluating a *global state predicate*. The global state predicate is a function that maps from a set of global states in the system to $\{true, false\}$. A *local predicate* is such a predicate whose truth value depends only on the state of a single process. The four well-known classes of global predicates are stable, observer-independent, linear, and semi-linear.

Stable predicates [2] are usually associated with distributed problems such as deadlock or termination. Once the system enters a state where the predicate is evaluated to *true*, it remains *true* also in all future states that the system can enter. On the other hand, when we monitor or debug an application, we are often interested in *non-stable* predicates. These predicates can evaluate to *true* only in certain states of the computation and they need not be evaluated to *true* in any future state.

Observer-independent predicates [1] are such predicates where *possibly* Φ and *definitely* Φ are equivalent. Any stable predicate is also observer-independent.

The definition of a *linear* predicate [12] is based on a "forbidden" state S (the predicate Φ in the cut containing state S must remain false until a successor to S is reached). A predicate is linear if for any cut, in which the predicate is false, at least one of the states is forbidden. A sub-class of linear predicates is a class of *regular* predicates. A predicate is regular if the set of global states that satisfy the predicate forms a sub-lattice of the global lattice.

The class of *semi-linear* predicates, first proposed in [12], contains all the three previous classes. Its definition is based on a "semi-forbidden" state S , which is irrelevant to the truth-value of the predicate. While we are looking for a cut where the predicate is true, we can disregard S in favor of its successor.

From another point of view, there are two more classes of predicates, *conjunctive* and *disjunctive* predicates. Their meaning is obvious.

3.1 Predicate Detection

Next, we examine the problem of finding out whether a *transitory state* occurred in an actual execution (a non-stable predicate was evaluated to true). This is what we require

when debugging a distributed system. The aim is to determine such points of the execution where a global state predicate was *definitely true* and the cases where it was *possibly true*.

Detection of a stable predicate is trivial. It is sufficient to detect it in any one consistent state of the system and we know for sure that it will remain true until the final state. Apart from that, the detection of non-stable predicates is more complicated.

Next, we describe operators *possibly*, *definitely* [4], *controllable*, and *invariant* [8]. For a predicate Φ in terms of linearizations L of the global history of the system's execution H .

Possibly Φ means that there is a consistent global state S through which L of H passes such that Φ is true in S . *Definitely* Φ means that for every L of H , there is a consistent global state S through which L passes such that Φ is true in S . *Controllable* Φ means that Φ is true in every consistent state along some L of H . *Invariant* Φ means that Φ is true in every consistent state along every L of H .

There are three approaches to the detection of global predicates. In the first one, global snapshot algorithm [2] is used repeatedly until a consistent state where predicate Φ holds is found. This approach can be used for detection of stable predicates only.

Secondly, the construction of a global state lattice [3] is used. The lattice captures the reachability relation between consistent global states of a distributed system. Nodes denote consistent global states and edges denote possible transitions between the nodes. This lattice is then explored and evaluated in order to detect stable and unstable predicates.

In the third approach, the whole lattice is not constructed. Instead only a subset of global states, based on the structure of the predicate, is identified.

In the *predicate detection* approach, we are only detecting whether a given predicate ever became true. Apart from that, the *predicate control* problem states that given a distributed computation and a global predicate, it is possible to add synchronization arrows (messages) to the computation such that the predicate always stays true.

3.2 Predicate Detection Algorithms

To our best knowledge, predicate detection algorithms have been designed for the above defined predicates, when p belongs to a specific predicate class. A review [8] of references to existing algorithms is summarized in Table 1.

Predicate p	Algorithm			
	EF(p)	AF(p)	EG(p)	AG(p)
conjunctive	[9]	[10]	[9]	[10]
disjunctive	[10]	[9]	[10]	[9]
stable	[2]	[1]	trivial	trivial
linear	[12]	open	[8]	[8]
observer-independent	[12, 1]	[12, 1]	[8]	[8]
regular	[17, 7]	open	[17, 7]	[17, 7]

Tab. 1: Predicate Detection Algorithms

4 Observation

In the world of observation, the detection of a global predicate is divided among a checker and non-checker processes [13]. The *non-checker processes* are at computation nodes and have the read-only access to local predicates and channels with incoming and outgoing

messages. The *checker process* is a process that determines whether these predicates are *true* in the same global state.

The non-checker processes monitor local predicates. These processes also maintain information about the send and receive channel history for all channels incident to them. The non-checker processes send a message to the checker process whenever the local predicate becomes *true* for the first time since the last program message was sent or received.

The checker process is responsible for searching for a consistent cut that satisfies the predicate. It considers a sequence of candidate cuts and eliminates those of them which are not either consistent or does not satisfy the predicate [4].

According to [11] there were three open problems in observation of distributed programs. As we found out later on, the first two problems have already been solved in [16]. Detecting exactly- k predicate and the detection of 2-local predicates are established to be NP-complete. Moreover, a polynomial-time algorithms for special cases of the problems have been presented there. Thus, the only remaining open problem to our best knowledge is the *detection of 2-SAT predicates*: Consider a boolean predicate q in CNF form. Detecting q is NP-complete, if each clause has three or more literals. If each clause has exactly one literal, then q can be detected efficiently using [14]. The question here is: What is the complexity of detecting 2-SAT predicates?

4.1 Monitoring Approaches

In this section we will discuss three different widely used approaches to monitoring. Before that, we should mention that there exists the *probe effect* [6] which can considerably affect any observation. Modifying the system in any way, for example, adding even a piece of code to a distributed program (or removing it from the code), may alter the timing in a distributed system.

Now let us get back to the three general approaches to monitoring a distributed system. *Hardware Monitoring* is performed by tailored devices that need to be adopted to a target system. Consequently, it is a rather expensive approach. However, these devices do not affect the system at all and so have no influence on the probe effect. Next, *Software Monitoring* is prone to the probe effect. However, it is not as expensive as hardware monitoring. In this case of monitoring, the probe effect may be avoided by allowing trace routines to remain inside the release version of a program [18]. Obviously, the remaining routines will cause performance degradation of the program. Finally, *Hybrid Monitors* try to avoid the disadvantages of both previous approaches. They have a hardware part as well as a software part, both of them are kept as small as possible.

Monitoring at different levels is not strictly comparable. It is likely that several levels of monitoring should be used to obtain a comprehensive picture of the monitored system.

5 Control

The next natural step after observation is control. In general, control is based on a notion of *supervisory process* [13]. Every process has associated with it the supervisory process. The supervisory process observes the underlying user process and controls it by delaying (or disabling) some events or by changing the order of incoming and outgoing messages.

The current programming methodology views processes as a simple execution of instructions. Such an execution can lead to a fault which could have been avoided if critical

events had been verified for their suitability before the execution. On contrary, human beings do not blindly follow instructions. They first observe and control the instructions before executing. Thus, the supervisory process can also be viewed as an auxiliary process that monitors (the observation) and adapts (the control) the underlying process to varying external behavior.

6 Replaying a Distributed Execution

We have provided a review of how a parallel system can be observed (monitored). In this section we will investigate issues of execution reproduction. Such issues are most often based on trace information recorded during the monitoring phase.

One of widely used approaches is to record an event history containing all the events generated by the program (*deterministic replay*). This history can be examined later on when the program has finished. Since the event history is often very large, some debuggers provide facilities to browse or query the history. Event histories can also be used to drive the program execution allowing the reproduction of erroneous computations.

Some systems can automatically check the history for suspicious behavior or transform a huge lower-level history of events into smaller and more suitable one. Instead of recording the entire history, the debugger can only take a snapshot of the program state and keep only that part of the history that follows the snapshot, it is so-called *incremental replay*.

The amount of information that must be recorded for each event depends on how the event history is going to be used [5]. There are three general levels of use that require increasing amount of information (detail) to be recorded for each event:

Browsing – The event history is examined through the use of specialized tools. Examination methods range from text editors to animations showing the state changes caused by particular events. Browsing requires only minimal information about each event.

Replay – The debugger uses the event history to control a re-execution of the program. This makes it possible to use debugging techniques such as breakpoints, state examination, and single stepping; without changing the behavior of the program.

Simulation – The event history can be used to simulate the surrounding environment of any single process. This enables the use of a sequential debugger on the process without re-executing the entire distributed program.

Replaying real-time systems has several additional problems. The external I/O and interrupts must be recorded in addition to the communications between processes. Moreover, since real-time systems have a time dependency, it may be important to simulate time during the replay (e.g. if behavior caused by timeouts takes place).

7 Conclusions

We have presented the state of the art in the area of the debugging long-running distributed and embedded applications. We investigated key issues for the debugging of such systems. These issues included exhaustive testing, the global state lattice, on-line/off-line observation and control, and incremental and deterministic replay. We pointed out that one of the three observation problems from [11] (detection of 2-SAT predicates) is still supposed to be open as well as searching for particular predicate detection algorithms from Table 1.

Our future work will be focused on studying the incremental replay in more detail and discover its possible improvements.

Support

Finally, I would like to thank to my supervisor Prof. Jiří Šafařík for his support and encouragement. The research was supported by the Ministry of Education of the Czech Republic, project no. MSM-235200005 – Information systems and Technologies.

Bibliography

1. Charron-Bost, B., Delporte-Gallet, C., Fauconnier, H.: Local and Temporal Predicates in Distributed Systems, *ACM Trans. on Prog. Languages and Systems*, 17(1):157–179, 1995.
2. Chandy, K. M., Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transactions on Computer Systems* 3:1, 195, pp.63–75, 1985.
3. Cooper, R., Marzullo, K.: Consistent Detection of Global Predicates, *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, published in *ACM SIGPLAN Notices*, 26(12):167–174, 1991.
4. Coulouris, G., Dollimore, J., Kindberg, T.: *Distributed Systems: Concepts and Design*, 3rd edition, 672 pages, Addison Wesley, ISBN 0-201-61918-0, 2000.
5. McDowell, Ch. E. – Helmbold, D. P.: *Debugging Concurrent Programs*, *ACM Computing surveys*, volume 21(4), pp.593–622, 1989.
6. Gait, J.: A Probe Effect in Concurrent Programs, *Source Software Practice & Experience archive* 16(3), pp.225–233, John Wiley & Sons, New York, USA, ISSN-0038-0644, 1986.
7. Garg, V. J., Mittal, N.: On Slicing a Distributed Computation, In *Proc. of the 15th Intl. Conference on Distributed Computing Systems (ICDCS)*, pp.322–329, 2001.
8. Garg, V. K., Sen, A.: *Detecting Temporal Logic Predicates on the Happened-Before Model*, Technical Report TR-PDS-2001-003, PDSL, ECE Dept. Univ. of Texas at Austin, 2001.
9. Garg, V. K., Waldecker, B.: Detection of Weak Unstable Predicates in Distributed Programs, *IEEE Trans. on Parallel and Distributed Systems*, 5(3):299-307, 1994.
10. Garg, V. K., Waldecker, B.: Detection of Strong Unstable Predicates in Distributed Programs, *IEEE Trans. on Parallel and Distributed Systems*, 7(12):1323-1333, 1996.
11. Garg, V. K.: Observation and Control for Debugging Distributed Computations, *Proc. 3rd Intl. Workshop on Aut. Debugging (AADEBUG'97)*, Linkoping, Sweden, pp.1–12, 1997.
12. Chase, C., Garg, V. K.: Detection of Global Predicates: Techniques and Their Limitations, *Distributed Computing*, 11(4):191–201, 1998.
13. Garg, V. K. – Tarafdar, A.: *Debugging in a Distributed World: Observation and Control*, 1998. <http://citeseer.ist.psu.edu/573151.html>
14. Garg, V. K. – Waldecker, B.: Detection of Weak Unstable Predicates in Distributed Programs, *IEEE Transactions on Parallel and Distributed Systems*, 5(3), pp.299–307, 1994.
15. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM* 21(7), pp.558-565, 1978.
16. Mittal, N. – Garg, V. K.: On Detecting Global Predicates in Distributed Computations, In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp.3–10, Phoenix, Arizona, USA, 2001.
17. Garg, V. J., Mittal, N.: Computation Slicing: Techniques and Theory, In *Proc. of the 15th Intl. Symposium on Distributed Computing (DISC)*, pp.78–92, 2001.
18. Thane, H. – Hansson, H.: Using Deterministic Replay for Debugging of Distributed Real-Time Systems, *EUROMICRO Conference on Real-Time Systems*, pp.265–272, IEEE Computer Society, 2000.