

# On Deterministic Replay

Jiří Patera \*

jipatera@kiv.zcu.cz

Jiří Šafařík \*

safarikj@kiv.zcu.cz

## Abstract

Today's programs in distributed and embedded systems are often designed for long-time running applications and, thus, they are 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 and controlling distributed programs.

## 1 Introduction

Many problems in distributed systems, especially in distributed debugging, can be viewed as special cases of the problem of observing and controlling a distributed computation.

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 lack of global state implies that computation of global functions introduces overhead in message-passing. This overhead can be reduced by restricting our attention to *state intervals* rather than states. A state interval is a sequence of states between two *external events* where an external event is the sending or receiving of a message, and also the beginning and termination of a process. For each state interval it is sufficient to record the extremal value of the monitored variable rather than all values assigned to the variable in the interval. The number of state intervals is usually considerably smaller than the total number of events in the system.

The presence of multiple processes substantially increases computation complexity. If there are  $n$  processes, the total number of possible global states is  $m^n$  where  $m$  is

---

\* Department of Computer Science and Engineering, University of West Bohemia in Pilsen, Univerzitní 8, 306 14 Plzeň, Czech Republic.

the number of state intervals of any process. This property is often referred to as a *combinatorial explosion*.

The outline of this paper is as follows. In Section 2, a technique of observing a global state of a distributed computation is investigated. It is followed by the discussion about three approaches to the monitoring. Next, an approach to controlling a distributed computation is studied in Section 3. Finally, deterministic replay and the Time Machine are explored in Section 4.

## 2 Observation

In the world of observation, the detection of a global predicate is divided among a checker and non-checker processes [4]. 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. Logical vector clock [7] (instead of real-time clock) are used to identify the moment when the predicate becomes *true*.

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.

According to [3] there were three open problems in observation of distributed programs. As we found out later on, the first and the second problems have already been solved in [8]. 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 [5]. The question here is: What is the complexity of detecting 2-SAT predicates?

### 2.1 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* [2] which can considerably affect any observations. Modifying the system in any way, for example, adding even a piece of code to a distributed program (or removing it from the program), may alter the timing in a distributed system.

In order to debug the system, we would like to put a probe into the internals of the program so that we could determine the cause of the problem. We usually do this by inserting auxiliary code that will monitor the system – e.g. non-checker’s code. This code will affect the system and if we are unlucky, it will do so in such a way that a bug will disappear during the observed executions of the system. And if we remove the probe (the non-checker process) later on, the bug may re-appear.

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, they do not affect the system at all and so have no influence on the probe effect. The quantities of hardware-layer messages and their relative size result in large amount of data. Another disadvantage is that they have to look at the very low level of information with low information content in comparison with the program execution.

**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 [10]. Obviously, the remaining routines will cause performance degradation of the program. Four architectural software monitoring approaches (kernel-probes, inline-probes, probe-tasks, and probe-nodes) are investigated in [9].

**Hybrid Monitoring** tries 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. This keeps the costs for the hardware part lower than in the case of pure hardware monitors. Furthermore, the probe effect is much smaller than in the case of pure software monitor.

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.

### 3 Control

The next natural step after observation is control. In general, control is based on a notion of *supervisory process* [4]. 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.

#### 3.1 On-line vs. Off-line Approaches

Debugging a distributed system is, in fact, the search for undesirable behavior and identification of its cause. In order to achieve this, the programmer needs to observe the program in a controlled environment so that they could set up exactly those conditions under which they want to test the program.

This can be done using one of the two possible approaches to the problems of observation and control – on-line or off-line debugging. When we speak of *on-line debugging*, we mean that a program is being debugged while it is running. As a consequence, we have no information on the future of the computation, we know only the past. On the contrary, we know everything about the computation when *off-line debugging* is used. The computation was monitored first and some trace information was recorded. So we know not only the past but also the future.

## 4 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.

### 4.1 Reproducibility and Completeness Problems

The *reproducibility problem* and the *completeness problem* are similar to each other in that both of them emerge only in non-deterministic distributed programs [6].

The reproducibility problem describes the fact that a certain behavior in a non-deterministic system cannot be repeated on command. Thus, it may be problematic to verify that a certain bug has been removed.

Confirmation that such a bug has been efficiently removed from the system can be identified by the *deterministic replay*. It is based on an event history of the execution where some particular failure occurred. The event history is made of significant events recorded at run-time and then is used off-line to reproduce and examine the system behavior. The examination can be of greater detail than the events recorded in the history. Nevertheless, it is not intended for speculative explorations of program behavior, since only previously recorded executions can be replayed.

Different invocations of a non-deterministic program can behave in different ways even though all controllable inputs are identical in all invocations. Testing the complete set of all possible combinations of input/output data and all execution orderings is normally referred to as *exhaustive testing*. Even in a small system the number of test cases can be extremely large and it increases drastically as the system grows. Therefore, exhaustive testing is normally not an option as it would require too long time to perform.

The alternative is to test only a subset of the input combinations, which leads to a certain level of confidence that the system specification was fulfilled.

### 4.2 Browsing, Replay, and Simulation

One of widely used approaches is to record an event history containing all the events generated by the program. 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.

If the history is complete enough, a single process can be debugged in isolation with the history providing the needed communication. If only the single suspect process is debugged with the rest of the program simulated by the event history, distributed debugging problem is reduced to that of debugging a single sequential process.

Finally, 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. If the interesting part of the execution can be identified, then the amount of information required for replay can be considerably reduced. 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.

This approach is often called *incremental replay*, only selected intervals of an execution are replayed without re-executing the whole program. This is useful especially in long-running distributed programs. Their execution can last for hours, days, or weeks. It is

unfeasible to restart it from the beginning in order to isolate a bug that manifested itself in a well-defined section of the program. To enable incremental replay, each process must periodically checkpoint (record) the state of its computation to a stable storage.

The amount of information that must be recorded for each event depends on how the event history is going to be used [1]. 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 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. Since real-time systems generally have a time dependency, it may also be important to simulate time during the replay (e.g. if behavior caused by timeouts takes place).

Note that if an executable is modified, either by re-compilation or re-linking, address references may change. Therefore, the current versions of programs (or their check-sums) should be recorded in the log in order to avoid failures caused by running improper program version at the nodes.

### 4.3 The Time Machine

The Time Machine is a debugging technique based on deterministic replay and presented in [11]. Trace information is recorded during run-time. This information includes interrupts, task-switches, timing, communication and data. The behavior of the whole system can then be deterministically reproduced and the system debugged both forward and backwards in time into a greater detail.

Only the data flow is recorded when debugging single-tasking real-time systems. To replay and debug multi-tasking real-time systems the control-flow has to be recorded in addition to the data-flow. One recorder has been enough until now. However, when debugging distributed real-time systems, one *recorder* at each node is needed. Afterwards, one *historian*, which derives a time-line, is run at each node. There are three basic elements in this debugging technique:

1. **The Recorder**, is a mechanism that collects all necessary information regarding task-switches, interrupts, and data.
2. **The Historian**, is the system that automatically analyzes and correlates events and data in the recording. Moreover, it composes these into a chronological time-line of breakpoints and predicates.
3. **The Actor**, deterministically replays the history in the debugger by generating interrupts, task-switches and restoring data as defined by the time-line.

Identification of any event is needed and it is achieved by saving when and where the event occurred. [11] use synchronized time base and time-stamping for distributed systems debugging. The location of the event is determined by using the *program counter* (*PC*). Since *PC* values can be revisited in loops, sub-routines, or recursive calls; recorded *PC* values are time-stamped (*TS*). The granularity of regular timers in CPUs make timestamps not so precise as they are sometimes required. Therefore, *instruction counter* (*IC*) can be used if it is supported by the target CPU. Finally, every event can be uniquely determined by a triplet  $\langle PC, IC, TS \rangle$ .

## 5 Conclusions

We have presented a 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 on-line/off-line observation and control, and incremental and deterministic replay. We pointed out that one of the three observation problems from [3] (detection of 2-SAT predicates) is still supposed to be open.

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

*The research was supported by the Ministry of Education of the Czech Republic, project no. MSM-235200005 – Information systems and Technologies.*

## Bibliography

1. McDowell, Ch. E. – Helmbold, D. P.: *Debugging Concurrent Programs*, ACM Computing surveys, volume 21(4), pp.593–622, 1989.
2. 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.
3. Garg, V. K.: *Observation and Control for Debugging Distributed Computations*, Proc. 3<sup>rd</sup> International Workshop on Automated Debugging (AADEBUG'97), Linkoping, Sweden, pp.1–12, 1997.
4. Garg, V. K. – Tarafdar, A.: *Debugging in a Distributed World: Observation and Control*, 1998. <http://citeseer.ist.psu.edu/573151.html>
5. 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.
6. Huselius, J.: *Debugging Parallel Systems: A State of the Art Report*, Mälardalen University, Department of Computer Science and Engineering, MRTC Report no. 63, 2002.
7. Lamport, L.: *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM 21(7), pp.558-565, 1978.
8. Mittal, N. – Garg, V. K.: *On Detecting Global Predicates in Distributed Computations*, In Proceedings of the 21<sup>st</sup> IEEE International Conference on Distributed Computing Systems (ICDCS), pp.3–10, Phoenix, Arizona, USA, 2001.
9. Thane, H.: *Monitoring, Testing and Debugging of Distributed Real-Time Systems*, PhD Thesis, Royal Institute of Technology, Sweden, 2000.
10. 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.
11. Thane, H. – Sundmark, D.: *Debugging Using Time Machines – Replay Your Embedded Systems History*, Real-Time & Embedded Computing Conference, Milan, Italy. 2001.