



## Distributed debugging techniques

Michel Adam, Michel Hurfin, Michel Raynal, Noël Plouzeau

► **To cite this version:**

Michel Adam, Michel Hurfin, Michel Raynal, Noël Plouzeau. Distributed debugging techniques. [Research Report] RR-1459, INRIA. 1991. <inria-00075102>

**HAL Id: inria-00075102**

**<https://hal.inria.fr/inria-00075102>**

Submitted on 24 May 2006

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# INRIA

UNITÉ DE RECHERCHE  
INRIA-RENNES

Institut National  
de Recherche  
en Informatique  
et en Automatique

Domaine de Voluceau  
Rocquencourt  
B.P. 105  
78153 Le Chesnay Cedex  
France  
Tel. (1) 39 63 55 11

## Rapports de Recherche

N° 1459

*Programme 1*  
*Architectures parallèles, Bases de données,*  
*Réseaux et Systèmes distribués*

### DISTRIBUTED DEBUGGING TECHNIQUES

Michel ADAM  
Michel HURFIN  
Michel RAYNAL  
Noël PLOUZEAU

Juin 1991





## 1.1 Proving it correct

When dealing with distributed software construction, one would like to follow the classical life cycle: specification, implementation, proof of correctness of the implementation with respect to the specification. This last step is highly desirable but extremely difficult to achieve, because the inherent non-determinism of interprocess communication

Two classes of proof techniques currently exist.

1. The first class deals with incremental program construction: successive refinements are performed, starting from the specification up to the final implementation. At every refinement step one has to prove that the previous implementation's properties are preserved by the new one. Axiomatic proof techniques also fit into this class[5].
2. The second class deals with *a posteriori* proofs: one must prove that all desired properties defined in the specification are met by the final implementation. Current automated proof techniques include the exhaustive construction of program's state graph which is then used to evaluate all the specification's predicates. If the state graph is finite then all possible behaviors of the program are taken into account by this technique and it is possible to check that a property is satisfied by every behavior. However, determining whether the graph is finite or not is undecidable in the general case. Moreover, the combinatorial explosion of the state graph size disallows the exhaustive construction of the state graph when dealing with middle or large scale programs.

When one is satisfied with verification of correctness on some behaviors only then partial simulation techniques are useful. The implementation is executed while some device observe the execution trace and checks that the current behavior complies with the specification. Of course nothing is proved, but the confidence in the program may be increased.

## 1.2 Definition of the distributed debugging activity

In its simplest form the observer checking for expected behavior is the programmer herself, who controls "by sight" that the behavior is an expected one. In other words, the programmer debugs the program. With a debugger a programmer can control the execution of the program (setting breakpoints) and can examine the state of the program then it has stopped. A distributed debugger has to offer the same facilities to debug a distributed program. Section 2 exposes the problems which appear when one tries to extend the fundamental concepts of sequential debugging to distributed contexts. Section 3 lays out basic solutions to these problems.

# 2 Fundamental problems

## 2.1 Observation problems

### 2.1.1 Global state

The state of a distributed program is made of the state of the processes taking part in the program and of every communication channels interconnecting these processes. Computing continuously this state is impossible because of the unknown communication delays; a class of distributed algorithms is able to compute *consistent* snapshots [4, 3] which are a good approximation of the

global state. Computing these snapshots is expensive with respect to CPU time and bandwidth. Hence the classical way of defining breakpoints as state predicates is not well suited to distributed debugging.

### 2.1.2 Events

Debugging a program requires knowledge of the successive states assumed by the program as well as information upon the events leading to the state transitions. Simply collecting these events and presenting them to the programmer is insufficient: in the general case ordering these events (e.g. determining causality relations) requires special techniques; the more detailed the information about event causality, the more expensive these techniques are.

### 2.1.3 Granularity

Every debugger needs a good definition of what a step of execution is. Sequential debuggers allow several granularities: one machine instruction step, one source line step or one procedure execution. Distributed debuggers have to encompass the parallel aspects of distributed program execution. Defining a granularity is not a trivial task, as bad choices may lead to difficulties in the debugger implementation.

## 2.2 Reproducibility

Typical debugging activities make use of a reproducibility facility: once an error is discovered careful reexecutions of the program are used to pin-point the context and the causes of the error. Conventional debugger rely implicitly on the deterministic behavior of sequential programs (when they do not use non-deterministic constructions [6]): the same input data gives the same program behavior. Unfortunately, common distributed programs are non-deterministic; indefinite transmission delays and non-deterministic constructs prevent these programs from being reproducible, unless special *ad hoc* techniques are implemented in the debugger. More details are given in section 3.

## 2.3 Halting the computation

When for some reason the computation has to be halted (e.g. a breakpoint condition is satisfied, an uncontrolled exception has occurred or the user requested a break), questions about bringing the computation into a consistent state arise: halting every process is not instantaneous and the computation must be at a point compatible with the debugging granularity.

# 3 Basic solutions

## 3.1 Definition of a model

As a distributed execution is more than a juxtaposition of sequential processes we have to define a model of distributed computation. All techniques given in the rest of the paper are defined within this model and are relevant as long as the distributed program satisfies the properties given in the model.

Our model is Lamport's one [11]. The observable actions from a process execution are: message sending, message reception, internal action. The last type of action allows us to define any granularity for process instructions; for instance we may decide that instructions other than inter-communication instructions are invisible to the debugging system. An event is an action occurrence. Remark that the concept of time (i.e. wall clock time) makes no sense in this model.

Let  $P_i$  be a process from the distributed program; the set of events occurring on this site is noted by:  $e_0^i . e_1^i \dots e_k^i \dots e_\infty^i$ , where  $e_0^i$  and  $e_\infty^i$  are pseudo-events named *initial event* and *terminal event* respectively. Let the order relation  $\prec_L$  be:

$$(e_l^i \prec_L e_m^j) \iff ((i = j) \wedge (l \leq m - 1))$$

Let the order relation  $\prec_C$  be:

$$(e_l^i \prec_C e_m^j) \iff \begin{cases} \text{There exists a message } m \text{ such that:} \\ e_l^i = \text{sending of } M \text{ to } P_j \\ e_m^j = \text{reception of } M \text{ from } P_j \end{cases}$$

The reflexo-transitive closure of  $\prec_L \cup \prec_C$  is noted  $\prec$ . Event  $e_m^j$  is causally dependent of  $e_l^i$  when  $e_l^i \prec e_m^j$ . Two events  $e_l^i$  and  $e_m^j$  are concurrent if and only if  $e_l^i \not\prec e_m^j \wedge e_m^j \not\prec e_l^i$ . To simplify the definitions given in the rest of this paper, the following assumptions are made:

$$\forall i \forall j e_0^i \prec e_1^j \tag{1}$$

$$\forall i \forall j e_{\infty-1}^i \prec e_\infty^j \tag{2}$$

## 3.2 Reproducibility

As knowing the very event history of an execution is impossible, the reproduction of this same execution is impossible: only an equivalent reexecution is feasible.

### 3.2.1 Equivalence of executions

Two executions  $E_1$  and  $E_2$  are considered equivalent if and only if:

- they have the same set of events,
- the order relations  $\prec_{L_1}$ ,  $\prec_{L_2}$  and  $\prec_{C_1}$ ,  $\prec_{C_2}$  are the same.

In other words, every process execute its actions in the same order in the two executions, and this implies that the order of reception of the messages are the same.

### 3.2.2 Reproduction techniques

The two causes of non-deterministic behavior are the local non-determinism (from non-deterministic statements, e.g. non-deterministic choice statement, random number generator and real time clocks) and the asynchronous interprocess communications (global non-determinism).

Recording all values computed by local non-deterministic statements allows locally deterministic reexecution of every process code.

Hence the order of occurrence of internal actions and message emission is then preserved.

Another technique is at need to ensure that the message receptions during the reexecution occur in the same order than during the initial execution [13, 12, 7]. Tagging the message with some information will ensure this.

Below is an informal description of these techniques.

- During the initial execution every process maintains a log file of locally occurring events:
  - For every process  $P_i$ , every message sent to  $P_j$  is tagged with tuple  $(i, s_i^j)$ , the  $s_i^j$  being a variable counting the number of messages sent by  $P_i$  to  $P_j$ .
  - Upon reception by  $P_j$  of a message tagged with tuple  $(i, s_i^j)$ , the tuple is recorded in  $P_j$ 's log.
  - When a process performs a local non-deterministic choice of statements, the result of the choice is recorded.
  - When a process reads a value from a non-deterministic source (timer, random number generator), the value is recorded.

During a program reexecution, every process takes care to behave deterministically by reading its log record.

- When a non-deterministic choice of statement is reached, the result of the choice is taken from the log.
- When a value from a non-deterministic source is to be read, the value returned is taken from the log.
- When a process  $P_j$  receives a message tagged with tuple  $(i, s_i^j)$ , this message is stored until its tuple equals the next reception tuple in  $P_j$ 's log file.

Recording every process actions induce an effect known as the *probe effect*: the non-deterministic behaviour of a program when the record system is on may be different from the same program's behaviour when no recording is performed, as timing are different.

### 3.3 Distributed breakpoints

#### 3.3.1 Local breakpoints

Common sequential debugging techniques make use of manual or automatic setting of breakpoints (automatic setting allows step by step execution) with machine instruction or source line granularity. Describing halting conditions with predicates on the state is also possible. Such features need to be considered in a distributed context[1].

Considering the cost of global state computation, we want to keep the possibility of defining breakpoints on any single process. Such breakpoints are named *local breakpoints*, as they do not mention other processes' states or events. One can specify them as predicates on the local state or as a trace of local events. More formally, a set of breakpoints made of event traces defines a language and well-known techniques of language specification and parsing are available [2].

### 3.3.2 Global breakpoints

Global breakpoints specify conditions over more than one process. A simple form of these is a disjunction of local breakpoints: the computation stops as soon as one local breakpoint condition is satisfied.

However, this form of breakpoint is not powerful enough to express frequent behaviors of a distributed computation: one is often interested in pin-pointing situations such as overtaking of specific messages (e.g. release indication received before request indication), etc. In this case no good technique is currently known.

### 3.3.3 Halting a program

Let us assume that the global breakpoint is the conjunction of  $k$  local breakpoints, for a set of  $N$  processes. When the global breakpoint occur,  $k$  processes stop because their local breakpoint conditions are satisfied and  $N - k$  processes have to be stopped in some state, which are necessarily visible actions (events) with respect to the computation model.

Let  $e_{s_i}^i$  be the event process  $P_i$  stops at. Let  $\phi$  be the place the distributed program stops at because of local breakpoints by

$$\phi = \{ e_{s_i}^i \mid 1 \leq i \leq N \}$$

Every event  $e_{s_i}^i$  which gives a local state satisfying the local breakpoint is determined; this set is noted  $\phi_B$ . The  $N - k$  other events forms a set  $\phi_C$ . The global breakpoint  $\phi = \phi_B \cup \phi_C$  is consistent if and only if every reception taken into account in  $\phi$  (i.e. every reception on some  $P_j$  occurring before or at  $e_{s_j}^j$ ) has its corresponding emission taken into account in  $\phi$ . More formally:

$$\forall e_x^i \forall e_y^j, e_y^j \prec_C e_x^i \wedge e_x^i \prec e_{s_i}^i \implies e_y^j \prec e_{s_j}^j$$

Any choice of event satisfying the above property is acceptable. We are going to detail two particular solutions. Let us first assume that only one local breakpoint condition is satisfied, i.e. process  $P_b$  is the only one to stop because of a local breakpoint:  $\phi_B = \{ e_{s_b}^b \}$ . We are now defining the  $N - 1$  other events with respect to  $e_{s_b}^b$ .

The *minimal state* is defined as

$$\phi_{\min}^b = \{ e_{\max\{k \mid e_k^i \prec e_{s_b}^b\}}^i \mid 1 \leq i \leq N \}$$

The *maximal state* is defined as

$$\phi_{\max}^b = \{ e_{\min\{k \mid e_{s_b}^b \prec e_k^i\} - 1}^i \mid 1 \leq i \leq N \}$$

**Theorem 1** *Both  $\phi_{\min}^b$  and  $\phi_{\max}^b$  are consistent.*

**Proof** For sake of brevity only  $\phi_{\min}^b$ 's consistency proof is given. We prove this theorem by contradiction. Let  $e_x^i$  and  $e_y^j$  be two events occurring on  $P_i$  and  $P_j$  respectively. The negation of the consistency definition is:

$$e_y^j \prec_C e_x^i \tag{3}$$

$$\wedge e_x^i \prec e_{\max\{k \mid e_k^i \prec e_{s_b}^b\}}^i \tag{4}$$

$$\wedge \neg(e_y^j \prec e_{\max\{k \mid e_k^j \prec e_{s_b}^b\}}^j) \tag{5}$$



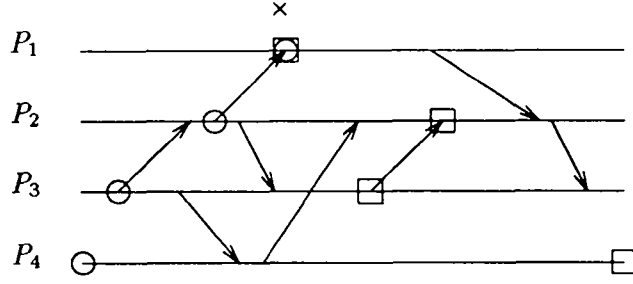


Figure 1 : Minimal and maximal states

Hypotheses (3) and (4) give

$$e_y^j \prec e_{\max\{k|e_k^j \prec e_{s_b}^b\}}^i \quad (6)$$

$$\vdash e_y^j \prec e_{s_b}^b \quad (7)$$

As term (5) refers to events occurring on the same process  $P_j$ , it is equivalent to

$$e_{\max\{k|e_k^j \prec e_{s_b}^b\}}^j \prec_L e_y^j \quad (8)$$

which implies

$$\max\{k|e_k^j \prec e_{s_b}^b\} < y \quad (9)$$

which contradicts (7). Thus the implication defining the consistency of a state is true.Q.E.D.

Figure 1 gives an example of minimal and maximal global states for a computation without internal actions. The minimal state is represented by circles and the maximal one by squares. The local breakpoint condition event is represented by a cross on  $P_1$ 's event log.

Let us now suppose that  $|\phi_B| = k$ , in other words that  $k$  local breakpoint conditions are satisfied. Without loss of generality let assume that these conditions occur on processes  $P_1, \dots, P_k$ . Let  $e_{s_1^1}^1, \dots, e_{s_k^k}^k$  be elements of  $\phi_{\min}^1, \dots, \phi_{\min}^k$ . As all events  $e_{s_1^1}^1, \dots, e_{s_k^k}^k$  occur on the same process  $P_i$ , they totally ordered by the  $\prec$  relation. The minimal state associated to  $\phi_B$  is then

$$\phi_{\min}^{[1,k]} = \{e_{s_i^x}^i \mid s_i^x = \max\{s_1^1, \dots, s_i^i, s_i^k\}, 1 \leq i \leq N\}$$

Similarly, the maximal state associated to  $\phi_B$  is

$$\phi_{\max}^{[1,k]} = \{e_{s_i^{x'}}^i \mid s_i^{x'} = \min\{s_i^{1'}, \dots, s_i^{k'}\}, 1 \leq i \leq N\}$$

**Theorem 2** Both states  $\phi_{\min}^{[1,k]}$  and  $\phi_{\max}^{[1,k]}$  are consistent.

**Proof** For brevity's sake we prove only that  $\phi_{\min}^{[1,k]}$  is consistent. Let  $e_x^i$  and  $e_y^j$  be two events occurring on  $P_i$  and  $P_j$  respectively. Let us assume that

$$e_y^j \prec_C e_x^i \quad (10)$$

$$\wedge e_x^i \prec e_{\max\{s_i^1, \dots, s_i^k\}} \quad (11)$$

Let  $s_i^l$  be  $\max\{s_i^1, \dots, s_i^k\}$ . In other words,  $e_{\max\{s_i^1, \dots, s_i^k\}}^i \in \phi_{\min}^l$ . Let  $e_{s_i^l}^j$  be the corresponding event on process  $P_j$ , i.e.  $e_{s_i^l}^j \in \phi_{\min}^l$ . As theorem 1 ensures that  $\phi_{\min}^l$  is consistent we have  $e_y^j \prec e_{s_i^l}^j$ . By construction we have

$$e_{s_i^l}^j \prec e_{\max\{s_j^1, \dots, s_j^k\}}^j$$

and thus

$$e_y^j \prec e_{\max\{s_j^1, \dots, s_j^k\}}^j$$

Q.E.D.

The concept of minimal state appeared in [8].

### 3.3.4 Implementation issues

An implementation of a distributed breakpoint mechanism must provide mechanisms to reset the global state to  $\phi_{\min}$  or  $\phi_{\max}$  when a breakpoint condition is detected. This problem is isomorphic to transaction rollback mechanisms in databases and to failure recovery techniques in distributed systems[10, 14, 9]. Common techniques rely on checkpointing: the local states are periodically recorded and the local actions of the processes since the last checkpoint are logged. The checkpointing policy is of utmost importance, since naive logging techniques are often inefficient.

## 4 Conclusion

The concepts exposed in this paper are fundamental for defining what a distributed debugger is. No particular theoretical difficulty prevents the reproducibility concept presented here from being implemented, but log size and cost are to be carefully considered. The simple breakpoint system exposed in this paper is also fairly easy to implement but suffer from limitations when one wishes to express global breakpoint conditions; expressing such conditions and implementing them is not currently mastered in the current state of art.

## Références

- [1] M. Ahuja, A.D. Kshemkalyani, and T. Carlson. A basic unit of computation in distributed systems. In *The 10<sup>th</sup> international conference on distributed computing systems*, pages 12–19, 1990.

- [2] F. Baiardi, N. De Francesco, and G. Vaglini. Development of a debugger for a concurrent language. *IEEE Transactions on Software Engineering*, SE-12(4):547–553, April 1986.
- [3] L. Bougé. Repeated snapshots in distributed systems with synchronous communications and their implementation in CSP. *Theor. Computer Science*, 49:145–169, 1987.
- [4] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 63–75, February 1985.
- [5] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley, 1988.
- [6] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [7] C.J. Fidge. *Dynamic Analysis of event ordering in message-passing systems*. PhD thesis, Australian National University, Departement of Computer Science, March 1989.
- [8] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *The 10<sup>th</sup> international conference on distributed computing systems*, pages 134–141, 1990.
- [9] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of algorithms*, 11:462–491, 1990.
- [10] R. Koo and Toueg S. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software engineering*, SE-13(1):23–31, 1987.
- [11] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [12] 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.
- [13] E. Leu, A. Schiper, and A. Zramdini. Efficient execution replay technique for distributed memory architectures. In *Proceedings of 2<sup>nd</sup> European Distributed Memory Computers Conference*, pages 315–324, LNCS 487, Springer Verlag, 1991.
- [14] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA  
1991

- PI 580 DESCRIPTION ET SIMULATION D'UN SYSTEME DE CONTROLE  
DE PASSAGE A NIVEAU EN SIGNAL  
Bruno DUTERTRE  
Mars 1991, 66 Pages.
- PI 581 THE SYNCHRONOUS APPROACH TO REACTIVE AND REAL-TIME  
SYSTEMS  
Albert BENVENISTE  
Avril 1991, 36 Pages.
- PI 582 PROGRAMMING REAL TIME APPLICATIONS WITH SIGNAL  
Paul LE GUERNIC, Michel LE BORGNE, Thierry GAUTIER  
Claude LE MAIRE  
Avril 1991, 36 Pages.
- PI 583 ELIMINATION OF REDUNDANCY FROM FUNCTIONS DEFINED  
BY SCHEMES  
Didier CAUCAL  
Avril 1991, 22 Pages.
- PI 584 TECHNIQUES POUR LA MISE AU POINT DE PROGRAMMES RE-  
PARTIS  
Michel ADAM, Michel HURFIN, Michel RAYNAL, Noël PLOUZEAU  
Mai 1991, 10 Pages.
- PI 585 TOWARDS THE CONSTRUCTION OF DISTRIBUTED DETECTION  
PROGRAMS, WITH AN APPLICATION TO DISTRIBUTED TERMINA-  
TION  
Jean-Michel HELARY  
Mai 1991, 24 pages.
- PI 586 OPAC : A COST-EFFECTIVE FLOATING-POINT COPROCESSOR  
André SEZNEC, Karl COURTEL  
Mai 1991, 26 Pages.
- PI 587 ON FAILURE DETECTION AND IDENTIFICATION : AN OPTIMUM  
ROBUST MIN-MAX APPROACH  
Elias WAHNON, Albert BENVENISTE  
Mai 1991, 24 Pages.
- PI 588 BOUNDED-MEMORY ALGORITHMS FOR VERIFICATION ON THE  
FLY  
Claude JARD, Thierry JERON  
Mai 1991, 14 pages.

ISSN 0249-6399