Distributed debugging techniques

Michel Adam, Michel Hurfin, Michel Raynal, Noël Plouzeau

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
Tél.:(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

RR_1459

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX
FRANCE
Téléphone : 99.36.20.00
Télex : UNIRISA 950 473F
Télécopie : 99.38.38.32

Techniques pour la mise au point de programmes répartis

Michel ADAM Michel HURFIN Noël PLOUZEAU
Michel RAYNAL

Projet ADP – Programme 1 – INRIA/IRISA
Campus de Beaulieu – 35042 RENNES CEDEX – FRANCE

Publication Interne n° 584 - Mai 1991 - 10 Pages

Résumé

Ce document présente certains problèmes fondamentaux auxquels un implémenteur est confronté lors du développement de metteurs au point pour des programmes fonctionnant sur des machines réparties. Les concepts classiques de la mise au point de programmes séquentiels sont en effet insuffisants. Après avoir défini l'état d'un programme principal, une équivalence d'exécution et le concept de reproductibilité d'un comportement de programme nous présentons quelques solutions simples à ces problèmes.

Distributed Debugging Techniques[1]

Abstract

This paper exposes fundamental problems that implementors of distributed debuggers have to cope with. Common sequential debugging concepts are not sufficient; we recall the current definitions of distributed computation state, execution equivalence, reproducibility of a behavior and breakpoints semantics and usage. Simple solutions for some problems are given.

1 Introduction

Debugging a program is generally a painful aspect of software development, but is also unavoidable when dealing with middle-scale and large scale software programs and systems. Nowadays building computers with many processing units is technically feasible and computer manufacturers provide us with such machines. However, it is obvious now that programming applications for these architectures is difficult, from the point of view of program correctness and efficiency.

In the rest of this paper, we will deal with aspects of distributed programs debugging, i.e. programs executing on parallel machines without shared memory and with message-based interprocess communication. Furthermore, we assume that the transmission delay of the messages exchanged between processes is arbitrary but finite.

[1]This work has been supported by the french GRECO C^3

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(U.R.A. 227)
UNIVERSITE DE RENNES I I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(UNITE DE RECHERCHE DE RENNES)

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 same 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 intercommunication 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: $e_0^i.e_1^i \ldots e_k^i \ldots e_\infty^i$, where e_0^i and e_∞^i are pseudo-events named *initial event* and *terminal event* respectively. Let the order relation $\prec_L$ be:

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

Let the order relation $\prec_C$ be:

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

The reflexo-transitive closure of $\prec_L \cup \prec_C$ is noted $\prec$. Event e_m^j is causally dependent of e_l^j 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 $e^i_{s_i}$ be the event process P_i stops at. Let ϕ be the place the distributed program stops at because of local breakpoints by

$$\phi = \{\, e^i_{s_i} \mid 1 \leq i \leq N \}$$

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

$$\forall e^i_x \forall e^j_y,\ e^j_y \prec_C e^i_x \wedge \epsilon^i_x \prec \epsilon^i_{s_i} \implies \epsilon^j_y \prec e^j_{s_i}$$

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: $\phi_B = \{e^b_{s_b}\}$. We are now defining the $N-1$ other events with respect to $e^b_{s_b}$.

The *minimal state* is defined as

$$\phi^b_{\min} = \{\, e^i_{\max\{k \mid \epsilon^i_k \prec \epsilon^b_{s_b}\}} \mid 1 \leq i \leq N \}$$

The *maximal state* is defined as

$$\phi^b_{\max} = \{e^i_{\min\{k \mid \epsilon^b_{s_b} \prec \epsilon^i_k\}-1} \mid 1 \leq i \leq N\}$$

Theorem 1 *Both $\phi^b_{\min}$ and $\phi^b_{\max}$ are consistent.*

Proof For sake of brevity only $\phi^b_{\min}$'s consistency proof is given. We prove this theorem by contradiction. Let e^i_x and e^j_y be two events occurring on P_i and P_j respectively. The negation of the consistency definition is:

$$\epsilon^j_y \prec_C \epsilon^i_x \tag{3}$$

$$\wedge \quad \epsilon^i_x \prec \epsilon^i_{\max\{k \mid \epsilon^i_k \prec \epsilon^b_{s_b}\}} \tag{4}$$

$$\wedge \quad \neg(\epsilon^j_y \prec \epsilon_{\max\{k \mid e^j_k \prec \epsilon^b_{s_b}\}}) \tag{5}$$

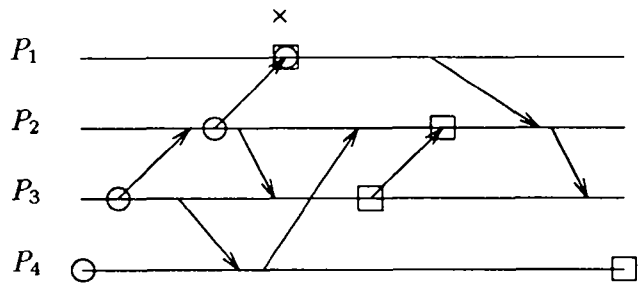

Figure 1 : Minimal and maximal states

Hypotheses (3) and (4) give

$$e^j_y \prec \epsilon^i_{\max\{k|\epsilon^i_k \prec \epsilon^b_{s_b}\}} \tag{6}$$

$$\vdash \quad e^j_y \prec c^b_{s_b} \tag{7}$$

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

$$e^j_{\max\{k|\epsilon^j_k \prec \epsilon^b_{s_b}\}} \prec_L e^j_y \tag{8}$$

which implies

$$\max\{k|\epsilon^j_k \prec \epsilon^b_{s_b}\} < y \tag{9}$$

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 $P_1, \ldots, P_k$. Let $e^i_{s^1_i}, \ldots, e^i_{s^k_i}$ be elements of $\phi^1_{\min}, \ldots, \phi^k_{\min}$. As all events $e^i_{s^1_i}, \ldots, e^i_{s^k_i}$ occur on the same process P_i, they totally ordered by the $\prec$ relation. The minimal state associated to ϕ_B is then

$$\phi^{[1,k]}_{\min} = \{e^i_{s^x_i} \mid s^x_i = \max\{s^1_i, \ldots, s^k_i\}, 1 \leq i \leq N\}$$

Similarly, the maximal state associated to φ_B is

$$\phi^{[1,k]}_{\max} = \{e^i_{s^{x'}_i} \mid s^{x'}_i = \min\{s^{1'}_i, \ldots, s^{k'}_i\}, 1 \leq i \leq N\}$$

Theorem 2 *Both states $\phi^{[1,k]}_{\min}$ and $\phi^{[1,k]}_{\max}$ 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

$$e_y^j \prec_C e_x^i \tag{10}$$

$$\wedge \quad e_x^i \prec e_{\max\{s_i^1,\ldots,s_i^k\}}, \tag{11}$$

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

$$e^j_{s_j^l} \prec e^j_{\max\{s_j^1,\ldots,s_j^k\}}$$

and thus

$$e_y^j \prec e^j_{\max\{s_j^1,\ldots,s_j^k\}}$$

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^{th} *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^{th} *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^{nd} *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 REPARTIS
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 TERMINATION
Jean-Michel HELARY
Mai 1991, 24 pages.

PI 586 OPAC : A COST-EFFECTIVE FOATING-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