Local states in distributed computations : a few relations and formulas

Eddy Fromentin, Michel Raynal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Local states in distributed computations: a few relations and formulas

Eddy Fromentin and Michel Raynal

N° 2192

Mars 1994

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

Local states in distributed computations: a few relations and formulas

Eddy Fromentin* and Michel Raynal*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Adp

Rapport de recherche n°2192 — Mars 1994 — 12 pages

Abstract: If events produced by processes of a distributed computation are generally supposed to be instantaneous, it is not the case for local states generated by these events. Due to message exchanges and synchronization local states have some duration. This paper defines notions about local states such as *weak precedence*, *strong precedence*, *weak concurrency* and *strong concurrency*. Moreover a few formulas based on vector clocks, and consequently usable in an operational context, are introduced to decide about relations between local states. These relations and formulas can be used either to debug, test or analyze distributed programs (especially for global properties detection) or to define consistent checkpoints.

Key-words: distributed computation, relevant event, causality, observation, weak precedence, weak concurrency, strong precedence, strong concurrency, debug, test, global property detection, checkpointing

(Résumé : tsvp)

This work has been supported in part by the Commission of European Communities under ESPRIT Programme BRA 6360 (BROADCAST), by the French CNRS under the grant Parallel Traces and by a French-Israeli grant on distributed computing.

*e-mail:{fromenti, raynal}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

États locaux dans les exécutions réparties: quelques relations et formules

Résumé : On considère généralement que les événements produits par une exécution répartie se produisent de façon instantanée, ce n'est pas le cas pour les états locaux produits par ces événements: ils perdurent de part les échanges de messages et la synchronisation. Ce rapport définit des relations sur les états locaux telles que les précédences *forte* et *faible* ainsi que les concurrences *forte* et *faible*. Nous donnons de plus un ensemble de formules permettant de déterminer quelles relations interviennent entre deux états locaux. Elles reposent sur l'utilisation de l'estampillage vectoriel et sont par là même directement utilisables à un niveau opérationnel. L'intérêt de ces relations et formules réside dans le déverminage, le test ou l'analyse des programmes répartis (en particulier pour le calcul de prédicats globaux), ou encore dans la définition de points de reprise.

Mots-clé : exécution répartie, événements observables, ordre causal, observation, précédence forte, précédence faible, concurrence forte, concurrence faible, déverminage, test, calcul de prédicats globaux, points de reprise

1 Introduction

Execution of a distributed program by a distributed system generates a set of *events* which is structured by a partial order relation usually called *happened before* or *causal precedence* [9]. Related events have been executed sequentially whereas unrelated events might have been executed concurrently (unrelated events are said to be *independent*). Lamport's clocks allows to timestamp events, with an integer clock, in a consistent way[1]. Fidge [4] and Mattern [11] proposed a clock system, based on vectors of integers, that allows to decide whether two events are related or not by the causal precedence relation.

An event produces a new local state for the process in which it occurs. Events are generally considered as instantaneous; consequently each local state lasts from the event that produced it till the next event issued by the same process. In this paper we are interested by relations on local states of a distributed computation. Several precedence and concurrency relations are introduced and simple formulas (based on vector clocks) are given to decide whether two local states are related or not. These formulas have a lot of uses, especially in the field of debug, test, analysis and checkpoint of distributed systems.

The paper is structured in the following way. Section 2 presents first different abstraction levels (called *Lamport's level* and *user's level*) that can be associated with a distributed computation. Section 3 then defines relations on local states (namely *strong* and *weak precedence* and *strong* and *weak concurrency*). Finally Section 4 gives simple formulas to decide whether two local states are related or not.

2 Events in a distributed computation

2.1 Distributed applications

A distributed application is composed of n sequential processes which communicate and synchronize by the only means of message passing. Such an application is executed by a distributed system; we suppose, for the sake of simplicity, there is one processor per process. Processors have local memories and channels allow them to exchange messages. There is neither shared memory nor a global clock; message transfer delays are finite but unpredictable.

[1] Let $C(e_1)$ and $C(e_2)$ be the timestamps associated with events respectively e_1 and e_2 by a clock C; C is consistent if: e_1 *causally precedes* $e_2 \Rightarrow C(e_1) < C(e_2)$.

2.2 Basic events and Lamport's level

Execution of a process P_i generates a sequence of *basic events*. Such an event either causes only changes to local variables (internal event) or involves communication (send or receive events). This sequence of basic events is usually called the history of P_i: $h_i = e_i^0 e_i^1 e_i^2 e_i^3 \cdots e_i^x \cdots$ where e_i^x is the x^{th} basic event produced by P_i; e_i^0 is a fictitious internal event that initializes P_i's context. Events are instantaneous.

Let H be the set of all basic events produced by the execution of a distributed application and $\stackrel{e}{\rightarrow}$ be the classical binary precedence relation on events defined by Lamport [9]:

$$e_i^x \stackrel{e}{\rightarrow} e_j^y \Leftrightarrow \begin{cases} & i = j \text{ and } x+1 = y \\ \textit{or} & \\ & e_i^x \text{ is the sending of a message and } e_i^y \text{ its reception} \\ \textit{or} & \\ & \exists e_k^z \text{ such that } e_i^x \stackrel{e}{\rightarrow} e_k^z \textit{ and } e_k^z \stackrel{e}{\rightarrow} e_j^y \end{cases}$$

These basic events define an observation level; we call it *Lamport's level*. All communication events belong to Lamport's observation level (Figure 1, where events are black and white circles and messages are arrows, represents a distributed computation in the classical space-time diagram).

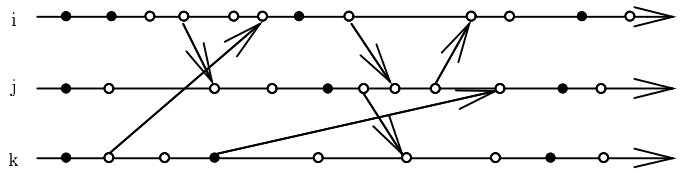

Figure 1: A distributed execution at Lamport's level (arrows indicates messages)

2.3 Relevant events and user's level

According to the user's needs only a subset of basic events are relevant (for example only changes to some variables are meaningful for the detection of a global predicate)[2].

[2] An other example is a checkpointing protocol where the only relevant events are the internal events that checkpoint local states [16].

Such events, composing a set R, are called *relevant* events. They define the *user's observation level*. Relation $\stackrel{e}{\rightarrow}$ structures R as a partial order. Interestingly the user's level inherits precedence between events induced by communication events, even when communication events are not defined as being relevant. (Figure 2 represents the space-time diagram of the computation displayed in Figure 1, at the user's level; relevant events are black circles).

Such a view of a distributed computation has been proposed by several authors. Marzullo and Sabel [10] use it to detect global predicates and Diehl *et al.* [3] use it to reduce the size of the lattice of global states in order to make efficient reachability analysis. Others authors have introduced notions such as *interval* [6, 7] or *spectrum* [14]; these notions adopt Lamport's level as the only observation's level of a distributed computation.

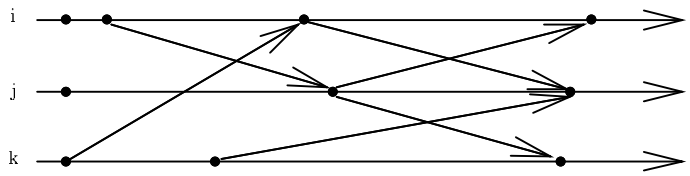

Figure 2: A distributed execution at user's level (arrows indicates causal precedence)

3 Local states of a distributed computation

3.1 Sequence of local states

Let R_i be the history of P_i at the user's level: $R_i = r_i^0 r_i^1 r_i^2 \cdots r_i^y \cdots$ (where r_i^y is the y^{th} relevant event of P_i; r_i^0 is the fictitious event that defines s_i^0 the initial local state of P_i). Event r_i^y $(y \geq 1)$ provoked P_i's local state change from s_i^{y-1} to s_i^y. So contrary to an event a local state lasts from the relevant event that produced it till the next relevant event issued by the same process. For notational convenience s_i^{y+1} (respt. r_i^{x+1}) will be denoted $next(s_i^y)$ (respt. $next(r_i^x)$).

s_i^{last} denotes the last local state of P_i. We suppose an additional fictitious event r_i^∞ that is executed once only when P_i is in local state s_i^{last} and produces local state $next(s_i^{last})$ identical to s_i^{last} as far as P_i's variables are concerned ($next(r_i^\infty)$ is not defined).

3.2 Strong precedence and weak concurrency

The set of local states generated by a distributed execution is partially ordered by a relation called *strong precedence*, denoted $\stackrel{s}{\rightarrow}$. Informally $s_i \stackrel{s}{\rightarrow} s_j$ means that s_i was no more existing when s_j began, as illustrated in Figure 3.

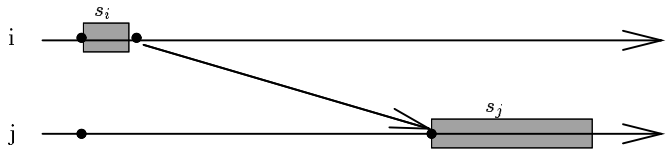

Figure 3: Strong precedence between local states

Formally, strong precedence is defined in the following way:

SP $\quad s_i^x \stackrel{s}{\rightarrow} s_j^y \stackrel{def}{\equiv} r_i^{x+1} \stackrel{e}{\rightarrow} r_i^y \ or \ s_j^y = next(s_i^x)$.

Two local states s_i and s_j are said to be *weakly concurrent*, denoted $s_i \parallel s_j$, if neither of them strongly precedes the other; such local states may have been simultaneous during the execution. More formally:

WC $\quad s_i \parallel s_j \stackrel{def}{\equiv} \neg(s_i \stackrel{s}{\rightarrow} s_j) \ and \ \neg(s_j \stackrel{s}{\rightarrow} s_i)$.

A global state Σ of the distributed execution is a n-uple of local states one from each process: $\Sigma = (s_1, s_2, \cdots, s_n)$. Such a global state is consistent if it could have been passed through by the distributed execution [1, 12], i.e.:

CGS $\quad \Sigma$ consistent $\Leftrightarrow \forall i \neq j : s_i \parallel s_j$

3.3 Weak precedence and strong concurrency

The set of local states is also structured by a weaker precedence relation, called *weak precedence* and denoted $\stackrel{w}{\rightarrow}$. Informally $s_i \stackrel{w}{\rightarrow} s_j$ if s_i began before s_j. In that case it is possible, if additionally $\neg(s_i \stackrel{s}{\rightarrow} s_j)$, that s_i and s_j might have existed simultaneously, as shown in Figure 4.

Formally relation $\stackrel{w}{\rightarrow}$ is defined in the following way:

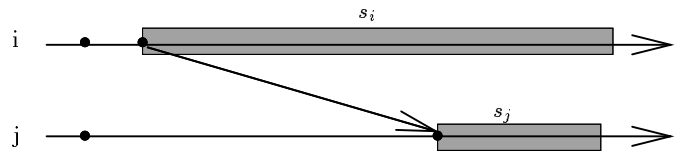

Figure 4: Weak precedence between local states

WP $s_i^x \overset{w}{\rightarrow} s_j^y \overset{def}{\equiv} r_i^x \overset{e}{\rightarrow} r_i^y$.

(R1): Of course: $s_i \overset{s}{\rightarrow} s_j \Rightarrow s_i \overset{w}{\rightarrow} s_j$.

Two local states s_i and s_j are said to be strongly concurrent, if neither of them began before the other. This strong concurrency relation is denoted $\|\!\|$.

SC $s_i \,\|\!\|\, s_j \overset{def}{\equiv} \neg(s_i \overset{w}{\rightarrow} s_j)\ and\ \neg(s_j \overset{w}{\rightarrow} s_i)$.

(R2): Definitions of concurrency are related by: $s_i \,\|\!\|\, s_j \Rightarrow s_i \parallel s_j$.

Moreover from **SP** and **WP** we can conclude:

(R3): $s_i \overset{w}{\rightarrow} s_j\ and\ \neg(s_i \overset{s}{\rightarrow} s_j) \Leftrightarrow s_i \overset{w}{\rightarrow} s_j\ and\ s_i \parallel s_j$

3.4 Remark

In [8] Lamport introduced two temporal precedence relations on operation executions. If A and B are two operations $A \longrightarrow B$ (read: A *precedes* B) means all actions composing A are completed before any action of B is begun; $A \dashrightarrow B$ (read: A *can affect* B) means some action of A precedes some action of B.

There is a similarity first between local states at Lamport's level and actions and second between local states at user's level and operations; with such a correspondence relations on local states at user's level $\overset{s}{\rightarrow}$ and $\overset{w}{\rightarrow}$ correspond to relations on operations $\longrightarrow$ and $\dashrightarrow$. (Remark that in our context a message reception affecting the receiver local state systematically entails a change to a new local state for the receiver process; so in our context $\dashrightarrow$, i.e. $\overset{w}{\rightarrow}$, is acyclic).

4 Detecting precedences

4.1 Vector clock

Sketched by several authors to solve particular problems [15], vector clocks have been formally introduced as a general mechanism by Fidge [4] and Mattern [11]. They constitute an operational device to encode causal precedence and concurrency of events. We use here such vector clocks with a slight modification: only relevant events are accounted. More precisely:

- $v_i[1 \cdots n]$ is the vector clock of process P_i; it is initialized to $\mathbb{1}_i$ (a zero vector with 1 in the i^{th} position).
- Each time P_i enters a new local state (or equivalently produces a new relevant event) $v_i[i]$ is incremented by 1. Moreover the current value of v_i constitutes the timestamp of the local state.
- Each message piggybacks the current value of the vector clock of its sender.
- When a message m, piggybacking $v(m)$, is delivered to a process P_i, v_i is updated to $max(v_i, v(m))$[3].

4.2 A few simple formulas

Let $v(s)$ be the timestamp associated to a local state s, P_i and P_j be two processes, and s_i (respt. s_j) be a local state of process P_i (respt. P_j). We have the following relations.

F1 $s_i \xrightarrow{s} s_j \Leftrightarrow v(s_i)[i] < v(s_j)[i]$

F2 $s_i \parallel s_j \Leftrightarrow v(s_i)[i] \geq v(s_j)[i]$ *and* $v(s_i)[j] \leq v(s_j)[j]$[4]

F3 $s_i \xrightarrow{w} s_j \Leftrightarrow v(s_i)[i] \leq v(s_j)[i]$

F4 $s_i \parallel\!\!\parallel s_j \Leftrightarrow v(s_i)[i] > v(s_j)[i]$ *and* $v(s_i)[j] < v(s_j)[j]$

F5 $s_i \xrightarrow{w} s_j \Rightarrow v(s_i)[j] < v(s_j)[j]$

F6 $s_i \parallel s_j$ *and* $s_i \xrightarrow{w} s_j \Leftrightarrow v(s_i)[i] = v(s_j)[i]$ *and* $v(s_i)[j] < v(s_j)[j]$

[3] $v_i := max(v_i, v(m)) \equiv \forall x \in 1 \cdots n : v_i[x] := max(v_i[x], v(m)[x])$.

[4] This relation has been also established in [1].

4.3 Proofs

In order to use results of [4, 11] we associate with each relevant event r_i the timestamp of the local state s_i produced by this event. As for local states, the timestamp of any relevant event r is denoted $v(r)$. In the following event r_i has been issued by P_i and produced local state s_i.

F1

$$\begin{array}{lll} s_i \xrightarrow{s} s_j & \Leftrightarrow next(r_i) \xrightarrow{e} r_j \ or \ next(r_i) = r_j & (\text{definition of } \xrightarrow{s}) \\ & \Leftrightarrow v(next(r_i))[i] \leq v(r_j)[i] & (\ [4, 11]) \\ & \Leftrightarrow v(r_i)[i] < v(r_j)[i] & (\text{as } v(next(r_i))[i] = v(r_i)[i] + 1) \\ & \Leftrightarrow v(s_i)[i] < v(s_j)[i] & \end{array}$$

F2

$s_i \parallel s_j \Leftrightarrow v(s_i)[i] \geq v(s_j)[i] \ and \ v(s_j)[j] \geq v(s_i)[j]$
Follows directly from the definition of the relation $\parallel$ and formula F1.

F3

$$\begin{array}{lll} s_i \xrightarrow{w} s_j & \Leftrightarrow r_i \xrightarrow{e} r_j & (\text{definition of } \xrightarrow{w}) \\ & \Leftrightarrow v(r_i)[i] \leq v(r_j)[i] & (\ [4, 11]) \\ & \Leftrightarrow v(s_i)[i] \leq v(s_j)[i] & \end{array}$$

F4

$s_i \parallel\!\parallel s_j \Leftrightarrow v(s_i)[i] > v(s_j)[i] \ and \ v(s_j)[j] > v(s_i)[j]$
Follows directly from the definition of $\parallel\!\parallel$ and formula F3.

F5

$$\begin{array}{ll} s_i \xrightarrow{w} s_j & \Rightarrow \neg(s_j \xrightarrow{w} s_i) \\ & \Rightarrow \neg(v(s_j)[j] \leq v(s_i)[j]) \\ & \Rightarrow v(s_i)[j] < v(s_j)[j] \end{array}$$

F6

$s_i \parallel s_j \ and \ s_i \xrightarrow{w} s_j \Leftrightarrow v(s_i)[i] = v(s_j)[i] \ and \ v(s_j)[j] > v(s_i)[j]$
Follows from the combination of F2, F3 and F5.

4.4 Inevitable global states

Notion of *inevitability* for global states has been introduced in [5]. All the inevitable global states are seen by all possible sequential observers[5] of a distributed computation. In others words a global state is inevitable if it is observer-independent. [5] proves the following necessary and sufficient condition for a global state to be inevitable:

$$\Sigma = (s_1, \cdots, s_i, \cdots, s_n) \text{ is inevitable } \Leftrightarrow \forall i, j : s_i \xrightarrow{w} next(s_j)$$

[5] A sequential observer of a distributed computation sees sequentially all the relevant events produced by this computation; and this sequence respects the partial order on events defined by the computation [1, 2, 12].

5 Conclusion

If events of a distributed computation can be considered as instantaneous, local states can not. A local state of a process lasts from a relevant event till the next one produced by the same process (which can be delayed due to synchronization reasons). Several precedence and concurrency relations have been defined on local states and associated decision formulas have been provided. These relations and formulas are particularly useful to debug, test, analyze distributed executions or to define consistent global checkpoints. Moreover if we are interested only by local states of a subset of processes, vector clocks can be shortened and have an entry only for processes of this subset; this can simplify detection of global properties that are on a subset of processes (see [13] for an example).

Acknowledgements

We are grateful to Ö. Babaoğlu and Cl. Jard for interesting discussions about notions of relevant events and observations. This work has been supported in part by the Commission of European Communities under ESPRIT Programme BRA 6360 (BROADCAST), by the French CNRS under the grant Parallel Traces and by a French-Israeli grant on distributed computing.

References

[1] Ö. Babaoğlu and K. Marzullo. *Consistent global states of distributed systems: fundamental concepts and mechanisms, in Distributed Systems*, chapter 4. *ACM Press, Frontier Series*, (S.J. Mullender Ed.), 1993.

[2] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, Santa Cruz, California, May 1991.

[3] C. Diehl, C. Jard, and J. X. Rampon. Reachability analysis on distributed executions. In *Theory and Practice of Software Development*, pages 629–643, TAPSOFT, Springer Verlag, LNCS 668 (Gaudel and Jouannaud editors), April 1993.

[4] J. Fidge. Timestamps in message passing systems that preserve the partial ordering. In *Proc. 11th Australian Computer Science Conference*, pages 55–66, February 1988.

[5] E. Fromentin and M. Raynal. *When all the observers of a distributed computation do agree.* Research Report 2194, INRIA, France, Mars 1994.

[6] V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Twelfth International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 253–264, Springer Verlag, LNCS 625, New Delhi, India, December 1992.

[7] A.P. Goldberg, A. Gopal, A. Lowry, and R. Strom. Restoring consistent global states of distributed computations. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 144–154, Santa Cruz, CA, May 1991. (Reprinted in SIGPLAN Notices, Dec. 1991).

[8] L Lamport. On interprocess communication. part 1: basic formalism. *Distributed Computing*, 1,2:77–85, 1986.

[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] K. Marzullo and L. Sabel. *Using Consistent Subcuts for Detecting Stable Properties.* Technical Report 91-1205, Dpt. of Computer Science, Cornell Univerity, Ithaca, New York, May 1991. 11 pages.

[11] F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robert, editors, *Parallel and Distributed Algorithms*, pages 215–226, North-Holland, October 1988.

[12] R. Schwarz and F. Mattern. *Detecting Causal Relationships in Distributed Computations : In Search of the Holy Grail.* To appear in Distributed Computing 7(4), 1994.

[13] A.I. Tomlinson and V.K. Garg. Detecting relational global predicates in distributed systems. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, California, May 1993.

[14] S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. In *Proc. of the Thirtieth Annual Allerton Conference*

on Communication, Control and Computing, pages 137–146, Urbana, Illinois (USA), October 1992.

[15] G.T. Wuu and A.J. Bernstein. Efficient solutions to the replicated log and dictionnary problems. In *Proc. 3rd ACM Symposium on PODC*, pages 233–242, 1984.

[16] J. Xu and R. H. B. Netzer. Adaptative independent checkpointing for reducing rollback propagation. In *Proc. 5th IEEE Symposium on Parallel and Distributed Processing*, pages 754–761, Dallas, December 1993.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399