I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 851

DEPENDENCY TRACKING AND FILTERING IN DISTRIBUTED COMPUTATIONS

CLAUDE JARD, GUY-VINCENT JOURDAN

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 84 71 71

Dependency tracking and filtering in distributed computations

Claude JARD, Guy-Vincent JOURDAN

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

Publication interne n ° 851 — Août 1994 — 15 pages

Abstract: The usual way for debugging a distributed program is to define a set of "observable events" among all events produced by the computation. Those events are sent to an observer process, which must check their correctness. It is well known that these events are only partially ordered by the "happened before" relation. There exists some coding which allows the observer to reconstruct the relation. We define three criteria to evaluate those coding and then propose a new coding which seems to offer a good compromise. The first criteria, called *intrusion*, measures the amount of additional information induced by the coding. The second criteria, called *filtering*, defines a set of events which have to be observed to make the coding correct. The third one, called *consistency*, defines the latency between the reception of an event by the observer and the moment at which it can compute the relation. It is thus a kind of measurement of the "on-line level" of the decoding process. We introduce a new coding, called *adaptive*, which provides filtering while keeping intrusion small.

Key-words: Distributed computations, Causal dependency, Observation, Partial orders.

(Résumé : tsvp)

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes

INRIA
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Capture et filtrage des dépendances dans les calculs répartis

Résumé : La façon ordinaire de déboguer un programme réparti est de définir un ensemble d'"événements observables" parmi tous les événements produits par le calcul. Ces événements sont envoyés vers un processus observateur qui doit décider de leur validité. Il est bien connu que ces événements observables ne sont ordonnés que partiellement par une relation de causalité. Il existe plusieurs codages qui permettent à l'observateur de reconstruire la relation de causalité au vu des événements observables. Nous définissons trois critères pour évaluer la pertinence de ces codages, et en profitons pour suggérer un nouveau codage nous semblant offrir un bon compromis. Le premier critère est l'*intrusion* qui mesure la quantité d'information rajoutée par le codage. Le second critère est le *filtrage*, qui est la capacité d'abstraire l'observation à n'importe quel sous-ensemble d'événements observables. Le dernier est la *vivacité* qui définit la latence entre la réception d'un événement par l'observateur et le moment auquel son ordre dans la relation peut être calculé : elle mesure l'habileté à travailler au vol. Nous introduisons un nouveau codage qui a la capacité de filtrage tout en assurant une faible intrusion.

Mots-clé : Calculs répartis, Dépendances causales, Observation, Ordres partiels.

1 Introduction

It is now established that the computations performed by distributed computing systems do not yield a linear sequence of events. The relationship between events inherently defines a partial ordering – concurrent events have no influence on one another.

In 1978, Lamport[9] defined the notion of dependency between events in distributed systems (called *causality*). Since then, considerable work has been done to retrieve these dependencies during execution. The problem of observing these dependencies occurs in many situations, such as designing, reasoning or debugging distributed programs. The usual techniques are based on timestamps associated to events and piggybacking of information on messages. Two major steps have been cleared ten years later by Fidge and Mattern [6, 10] with their famous "vector clock" mechanism, and by Zwaenepoel et al. [8, 7] for their notion of "direct dependency". These algorithms allow to code exactly Lamport's *happened before* relation using integers. Since then, the problem of the size optimality of such a coding has been raised [11, 3, 4, 1]. Unfortunately, as often in this case, there is not yet a consensus on the significant set of criteria nor on the minimal assumptions. In particular, are the codings able to manage abstraction of events ? This is of practical interest but not well understood yet.

In this paper, we analyze the problem of coding the *happened before* relation using the partially ordered set theory. We discuss the ability of the codings to filter irrelevant events or to resist to a disordered observation. We figure out that the direct dependency method actually requires some conditions on the observed events. We provide a new coding method which generalizes the direct dependency while providing abstraction features, and we give some experimental results.

2 Dependency tracking

We model a distributed system as a collection of processes, say $P = P_1 ... P_n$. The number of processes involved is not known. Each process executes its own

sequential program and interacts with the others by message exchanges[1]. We can view the execution of each process P_i in isolation as a sequence of events, corresponding to the state changes that take place in the process. This defines a total order $\widetilde{E_i} = (E_i, <_{E_i})$, where E_i is the set of events and the relation $<_{E_i}$ is just the time. The proper granularity defining the *internal events* varies from application to application. In addition to these internal events (I), we must consider the *communication events* (C), which partially synchronize the internal events. Following Lamport's definition of the *happened before* relation, we say that event x *directly happened before* event y, denoted by $x <_d y$, if

(1) x and y are events in the same process and x precedes immediately y; or

(2) x is the sending of a message m and y is the receiving of m.

Birth and death of processes can also be modeled by the *happened before* relation, provided that a child process is started by the reception of the first message from its father, and that its death is the sending of its last message to its father.

The transitive closure of the $<_d$ relation is the *happened before* relation $<_{I \cup C}$. A partial order on a set X will be denoted by $\widetilde{X} = (X, <_X)$. For $x \in I$, we note $\downarrow_I x$ the set of all predecessors of x in $\widetilde{I}$, x excluded, and $\downarrow_I^{im} x$ is the set of all immediate predecessors of x in $\widetilde{I}$. A suborder $\widetilde{O} = (O, <_O)$ induced by an order $\widetilde{I} = (I, <_I)$ is an order such that $O \subseteq I$ and $\forall x, y \in O, x <_O y$ iff $x <_I y$.

So far, there exist two major ways to track the dependency information:

- *Transitive dependency tracking* [13, 10, 12, 5]: complete *happened before* information is tracked. That is, the set of predecessors are accumulated upon the different receptions.
- *Direct dependency tracking* [8]: processes do not learn of dependencies resulting from chains of two or more messages.

[1] For the sake of simplicity, we present here the "standard" model of asynchronous message passing. There is no difficulty to use message passing to model synchronous communication.

In the following, we present two existing algorithms for tracking dependencies between an arbitrary subset O of internal events (called *observable events*). They are described in a uniform manner and have been slightly extended to support the notion of observable events.

O is distributed on the different processes: the set of observable events occurring on process P_j is denoted by O_j.

Transitive tracking and vector timestamps : this method [5] is based on a management of local counters piggybacked upon interaction. Loosely speaking, the principle of the method is to associate to each observed event x the set of its predecessors $\downarrow_O x$.

The set $\downarrow_O x$ can be coded at run time by a list $\mathcal{V}_x$ whose j^{th} component contains the number of predecessors of x on process P_j : $\mathcal{V}_x.j = |\downarrow_O x \cap O_j|$. Property 1 shows how to retrieve the information from the coding ($site(y)$ is the name of the process $P_{site(y)}$ on which y actually occurs) :

Property 1 $\forall x, y \in O,\ y \in V_x \Leftrightarrow \mathcal{V}_y.site(y) < \mathcal{V}_x.site(y)$

The Fidge's algorithm (see Figure 1 for an illustration) :

- each process P_i maintains a list L_i, initialized to $L_i.i = 0$,
- when an event x is observed : $\mathcal{V}_x := L_i; L_i.i := L_i.i + 1$,
- upon sending a message to P_j : L_i is piggybacked towards P_j,
- upon receiving from P_j : $L_i := max(L_i, L_j)$[2] where L_j is the piggybacked list from P_j.

The major problem with this method is the size of the lists L_i and $\mathcal{V}_x$, which can be equal to the number of processes created during the execution. That means that if there is a lot of births and deaths of processes, the size of the lists continuously grows, even if the number of processes simultaneously "alive" is always small.

[2] max is performed component by component.

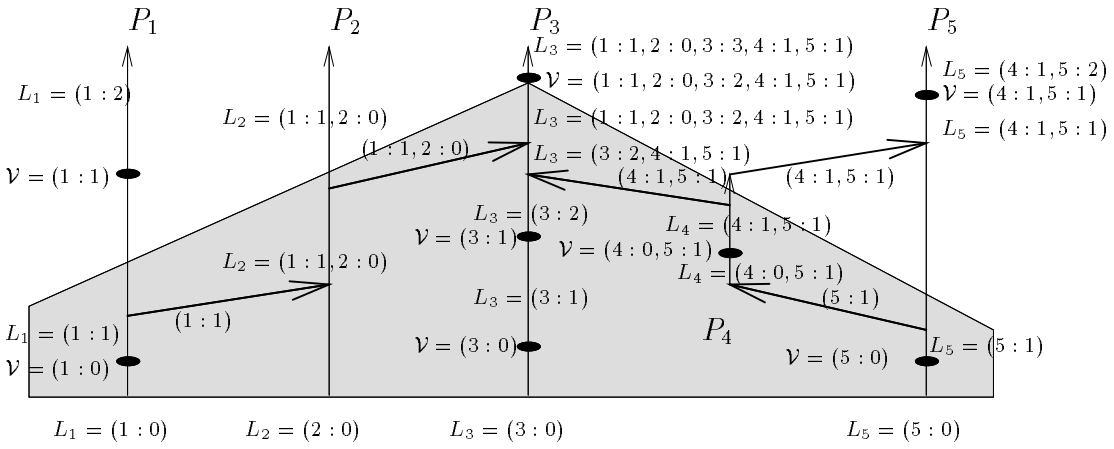

Figure 1: *Vector timestamps: black dots represent the observed events. The grey part shows $\mathcal{V}_x$, x being the third observed event of process P_3*

Direct dependency tracking and scalar timestamps : The idea of the Direct Dependency coding of Zwaenepoel et al.[8, 7] is to have on each site P_i a list whose j^{th} component is the number of events which occurred on P_j before the last interaction from P_j to P_i. Using our notations, we associate to each observed event x the set $D_x = \{y \in O : y \lhd x\}$ where $\forall x \in O_i, \forall y \in O_j, y \lhd x$ iff $i = j$ and $y <_i x$ or $\exists x', y'$ (y' is the sending of a message on P_j, x' the corresponding receipt on P_i) such that $y <_j y'$ and $x' <_i x$.

As previously, we can code the set D_x by a list of integers, whose i^{th} component contains the number of the direct predecessors of x on process P_i: $\mathcal{D}_x.i = |\{y \in O_i : y \lhd x\}|$. Property 2 shows how to retrieve the information from the coding :

Property 2 $\forall x, y \in O,\ y \in D_x \Leftrightarrow \mathcal{D}_y.site(y) < \mathcal{D}_x.site(y)$

Proof: $y \in D_x \Leftrightarrow y \lhd x \Leftrightarrow \{z \in O_{site(y)}, z \lhd y\} \subset \{z \in O_{site(y)}, z \lhd x\} \Leftrightarrow \mathcal{D}_y.site(y) < \mathcal{D}_x.site(y)$ ■

The (extended) Zwaenepoel's algorithm (see Figure 2 for an illustration) :

- each process P_i maintains a list L_i, initialized to $L_i.i = 0$,
- when an event x is observed : $\mathcal{D}_x := L_i; L_i.i := L_i.i + 1$,
- upon sending a message to P_j : $L_i.i$ is piggybacked towards P_j,
- upon receiving from P_j : $L_i.j := max(L_i.j, L_j.j)$.

The algorithm shows that we just have to piggyback a simple scalar, so this method offers a weak and bounded intrusion.

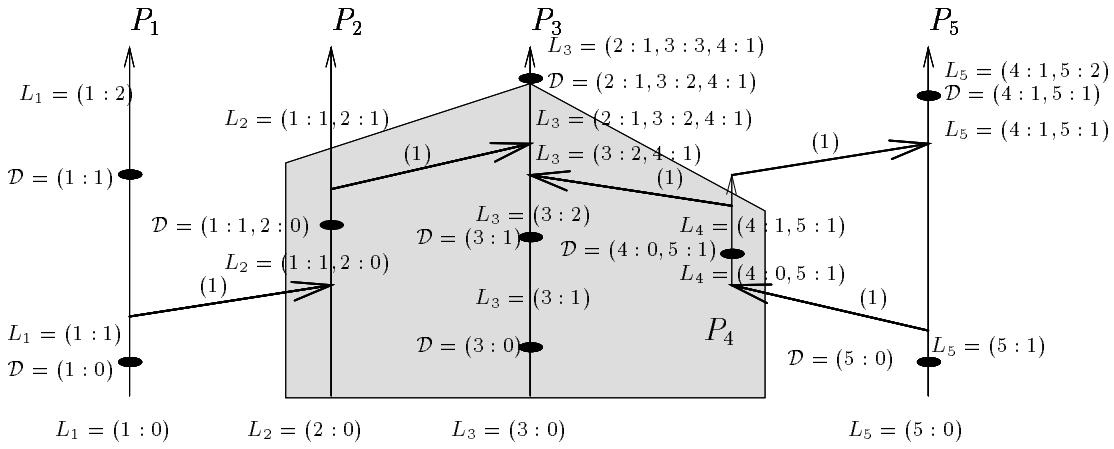

Figure 2: *Direct dependence timestamps*

3 Events filtering

Observing all the internal and communication events is not desirable, especially when the dependency information is to be traced for debugging purpose. Anyway, only relevant events have to be counted; and this is precisely the service offered by timestamps, instead of learning the entire space-time diagram.

But the ability of forgetting events depends on the tracking technique. Let $O \subseteq I$ be the subset of observable events which are actually observed. Our aim is to code the suborder $\widetilde{O} = (O, <_O)$ induced by $\widetilde{I}$. Are we free to consider any subset of I ? If positive, we say the coding offers a *strong filtering.*

For debugging purpose, the set of observed events can be sent to an observer process, but the actual ordering of observed events in front of the observer may be arbitrary. At a given moment, the observer has already received a subset H of O. What subset H of O must known by the observer to compute the suborder $\widetilde{H}$ induced by $\widetilde{O}$? If any subset is suitable for the computation, we say the coding offers a *strong consistency.*

Transitive dependency : we saw in the previous section that the vector timestamps compute the set $\downarrow_O x$ for any observable event x. Once such a set is known, it becomes straightforward to compute the *happened before* relation on any subset of observed events, the set of observed events being itself any subset of the set of observable events, as expressed by the Property 3. This shows that the Vector Timestamps method offers strong filtering and strong consistency.

Property 3 $\forall O \subseteq I, \forall H \subseteq O, \forall x \in H,\ \downarrow_H^{im} x \subseteq \downarrow_O x \cap H \subseteq \downarrow_H x$

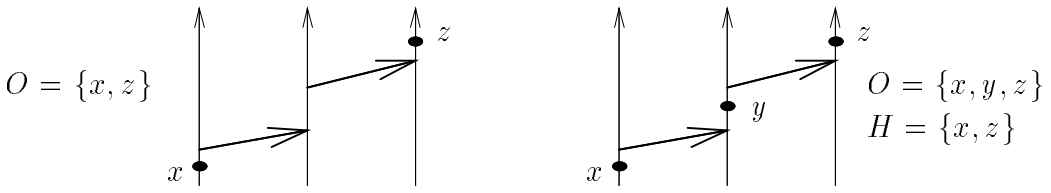

Figure 3: *(a) (no strong filtering) and (b) (no strong consistency)*

Direct dependency : unfortunately, this method does not ensure strong filtering: in Figure 3-a, there is no way to detect $x <_O z$ using D_x and D_z. To correct this problem, we have to add an assumption, called "NIVI" for Non InVisible Interaction, which states that after an interaction from P_j to P_i, we

must have an observed event on P_i before any interaction from P_i to any other process P_k.

In Figures 3-b, the NIVI condition is achieved. But still, an observer can not conclude $x <_o z$ before the reception of y. This reveals a weak consistency : to compute the order $\widetilde{H}$, the set H must be downward closed in $\widetilde{O}$ (also called an *IDEAL* of $\widetilde{O}$). Formally,
$\forall x \in H, \forall y \in O, y <_O x$ implies $y \in H$. This does not imply that the observer must receive the events according to a causal observation but that he cannot integrate an event x into the order $\widetilde{H}$ before the reception of all predecessors of x. Now, we can state :

Property 4 $\forall O \subseteq I : NIVI(O), \forall H \subseteq O : IDEAL(H), \forall x \in H, \downarrow_H^{im} x \subseteq D_x \subseteq \downarrow_H x$

Proof: Since H is an ideal of $\widetilde{O}$,we have $D_x \subseteq H$ and $\downarrow_H x = \downarrow_O x$. The property is then a consequence of the NIVI condition. ■

4 Adaptive timestamps

The idea of our coding is to combine the advantages of the two preceding methods. We want to obtain a weak intrusion while keeping the ability of abstraction. For that purpose, we note that the vector technique always propagates the whole information, the list L_i. If we accept a weak consistency, this is not necessary, as shows the Direct Dependency method. On the other hand, the Direct Dependency method propagates only the information about one process, the scalar $L_i.i$. This is not enough for a strong filtering, since this information can be lost. In fact, the information must be propagated while no observed event occurs. When such event occurs, the propagation can stop, since the information can be associated to that event, and then can be retrieved through it. Finally, we also figure out that these rules can be applied as well for a propagation inside a process: we do not have to record a dependency if a previous event on the process already records that dependency.

Using our formalism, we associate to each observed event x a set
$M_x = \{y \in O : y \ll x\}$, where the relation $\ll$, called *pseudo-direct* relation,

means the existence of a path of interactions from P_j to P_i, starting after y on P_j and ending before x on P_i, without any observed events after leaving P_j. Formally, $\forall x \in O_i, \forall y \in O_j, y \ll x$ iff $i = j$ and $y <_i x$ or $\exists s_1, r_1, s_2, r_2, \ldots, s_k, r_k$, such that s_i is the sending of a message from $P_{site(s_i)}$ to $P_{site(r_i)}$ and r_i the corresponding receipt on $P_{site(r_i)}$ and y happened before s_1 on P_j $(= P_{site(s_1)})$, r_l *directly* happened before s_{l+1} on $P_{site(r_l)}$ $(= P_{site(s_{l+1})})$ and s_k *directly* happened before x on P_i $(= P_{site(s_k)})$.

Remark 1 If $y \ll x$ then $\forall z \in site(y), z <_{site(y)} y \Rightarrow z \ll x$.

The Property 5 shows that our adaptive timestamping offers strong filtering and weak consistency :

Property 5 $\forall O \subseteq I, \forall H \subseteq O : IDEAL(H),\ \downarrow_H^{im} x \subseteq M_x \subseteq \downarrow_H x$

Proof: This is a simple consequence of the fact that H is an ideal and that if $y \in \downarrow_O^{im} x$, then there is a path without observed event between y and x. ■

The set M_x can be coded as usual by a list of integers, whose i^{th} component contains the number of pseudo-direct predecessors of x on process P_i: $\mathcal{M}_x.i = |\{y \in O_i : y \ll x\}|$. Proposition 6 shows how to retrieve the information from the coding :

Property 6 $\forall x, y \in O,\ y \in M_x \Leftrightarrow \mathcal{M}_y.site(y) < \mathcal{M}_x.site(y)$

Proof: Let $y \in M_x$, then $y \ll x$. If $site(x) = site(y)$, then $\mathcal{M}_y.site(y) < \mathcal{M}_x.site(y)$. Otherwise, there is a path of interactions from $P_{site(y)}$ to $P_{site(x)}$ which "arrives" on $P_{site(x)}$ before the occurrence of x and so by definition of $\ll$ (cf Remark 1)
$\{t \in P_{site(y)}, t \ll y\} \subset \{t \in P_{site(y)}, t \ll z\}$, i.e. $\mathcal{M}_y.site(y) < \mathcal{M}_z.site(y)$.

If $\mathcal{M}_y.site(y) < \mathcal{M}_x.site(y)$, then $\{t \in P_{site(y)}, t \ll y\} \subset \{t \in P_{site(y)}, t \ll z\}$, which implies $y \ll x$ (Remark 1). ■

The final algorithm is then derived (see Figure 4 for an illustration) :

- each process P_i maintains a list L_i, initialized to $L_i.i = 0$,

- when an event x is observed :
  $\mathcal{M}_x := L_i; L_i := L_i.i$; /* reset */
  $L_i.i := L_i.i + 1$;
- upon sending a message to P_j : L_i is piggybacked towards P_j,
- upon reception from P_j : $L_i := max(L_i, L_j)$

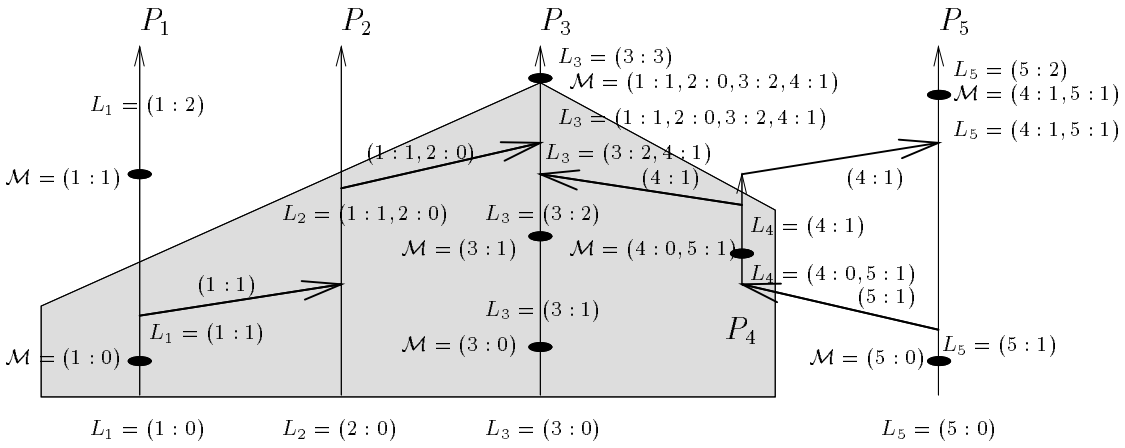

Figure 4: *Adaptive timestamps*

The size of the lists L_i and $\mathcal{M}_x$ depends on the number of processes traversed without encountering any observed events. We believe it offers a pragmatic *mean intrusion* (see the experiments below).

Bounding the size of the lists at run-time : our method offers an additional service: suppose we do not want to piggyback more than k integers ($k \geq 1$). So far, the size of the lists can be bigger – when there is a path going through more than k processes without observed events. So we add the following test : upon reception, after the max operation, if the size of the list L_i becomes bigger than k then we automatically generate a "null" observed

event, which resets the size to 1. The number of such null observed events generated at run time depends on the program and on the set of observable events, but we can expect this number to be much smaller than the number of observable events.

A few words about processing done by the observer : when an observer receives a new event x, he first has to check whether or not he has received all predecessors of x, using the list $\mathcal{M}_x$ (i.e. for all components j of $\mathcal{M}_x$, he has received the $\mathcal{M}_x.j^{th}$ element of P_j and all predecessors of this element have been received). If he does, then he computes the predecessors of x using Property 6. Otherwise, the computation is delayed while there is some predecessor missing.

Experiments : we have experimented with the different methods on a real example: the kernel of Jacobi's Algorithm. The matrix had a size of 100x100. It was distributed over 10 processes. Each process "owns" 10 columns of the matrix. Execution runs in a Single Program Multiple Data mode. In the first experiment, we observed all assignment of matrix elements on all process, i.e. 9604 events. The direct dependency technique has been used since the SPMD scheme fulfills the $NIVI$ condition. In the second experiment, we have only observed assignments on odd processes (4802 events). In this experiment, we had to use the adaptive technique since the observation does not check the $NIVI$ condition. Figure 5 summarizes the experimental results (size of stamps and intrusion are given by the size of the lists of integers). The decimal numbers are due to variations of the length of lists during the execution.

5 Concluding remarks

Dependency tracking is the basis of numerous distributed algorithms in distributed systems. Different specific methods have been designed to deal with the recovery problem [8, 12, 13]. Independently, different researchers enlighted fundamental concepts based on causal dependencies like global states [2] or lattice of consistent cuts [10, 3].

From an implementation point of view, these algorithms are all based on a particular coding of dependencies by list or vector of integers. The existence of several such codings in the literature ([10, 5, 4, 8], denoted by transitive,

	all assignments observed		*odd processes assignments observed*	
	Vector stamping	*Direct stamping*	*Vector stamping*	*Adaptive stamping*
average size of events' stamps (max. possible: 10)	8.9	1.1	8.86	1.24
average size of intrusion on messages (max. possible: 10)	9.83	1	9.83	1.94

Figure 5: *Experimental results*

direct or interval dependency) rises the problem of their comparison through their properties.

Recently, Wang at al. [14] has proved that direct dependencies have enough power to compute global checkpoints, though they do not capture all the causal relations.

In order to formally reason on this matter, the partial order theory provides a good mathematical tool, as exemplified in [3, 1, 4, 14].

We pursue this formalization by enlighting some properties of dependency codings. Particularly, we focus on the problem of events abstraction, that is the ability to compute suborders of the causality order. We have shown that the formal explanations yield naturally to other schemes of coding.

We get a new scheme which offers *strong filtering* as the Fidge's and Mattern's vector clocks, but with practical lower intrusion.

Figure 6 summarizes the evaluation of these methods. Of course, the best method would have strong filtering and consistency, and weak intrusion. But it seems to us that strong consistency and weak intrusion are mutually exclusive.

	Filtering		*Consistency*		*Intrusion*		
	Strong	*Weak*	*Strong*	*Weak*	*Strong*	*Medium*	*Weak*
Vector Timestamping	X		X		X		
Direct Dependencies		X		X			X
Adaptive Timestamping	X			X		X	

Figure 6: *Properties of coding*

References

[1] P. Baldy, H. Dicky, R. Medina, M. Morvan, and JF. Vilarem. *Efficient reconstruction of the causal relationship in distributed computations.* Technical Report 92-013, LIRMM, June 1992.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM TOCS*, 3(1):63–75, 1985.

[3] B. Charron–Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, July 1991.

[4] C. Diehl and C. Jard. Interval approximations of message causality in distributed executions. In Finkel and Jantzen, editors, *STACS*, pages 363–374, Springer–Verlag, LNCS 577, Cachan, February 1992.

[5] C.J. Fidge. A simple run–time concurrency measure. In *Proceedings of the 3^{rd} Australian Transputer and ACCAM User Group Conference*, pages 92–101, 1990.

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

[7] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In 10^{th} *IEEE International Conference on Distributed Computing Systems*, pages 134–141, 1990.

[8] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11:462–491, 1990.

[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] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms Bonas, France*, North Holland, September 1988.

[11] M. Singhal and A. Kshemkalyani. *An efficient implementation of vector clocks*. Technical Report, Department of Computer and Information Science, Ohio State University, October 1990.

[12] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. *ACM SIGACT-SIGOPS, Symp. Principles of Distributed Computing*, 223–238, 1989.

[13] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[14] Y.M. Wang, A. Lowry, and W.K. Fuchs. Consistent global checkpoints based on direct dependency tracking. *Information Processing Letters*, (50):223–230, June 1994.