k-Dependency Vectors: A Scalable Causality-Tracking Protocol *

Roberto Baldoni
Dipartimento di Informatica e Sistemistica
Università "La Sapienza"
Via Salaria 113
00198 Roma, Italy
baldoni@dis.uniroma1.it

Giovanna Melideo
Dipartimento di Informatica
Università degli Studi dell'Aquila
Via Vetoio, loc. Coppito
67010 L'Aquila, Italy
melideo@di.univaq.it

Abstract

In this paper we present a scalable causality-tracking protocol, called k-Dependency Vectors, which piggybacks on each application message a constant number k of integers (with $k \leq n$). These integers are selected from a vector of size n which is local at each process. By reducing the size of the piggybacked information, only a subset of the causal dependencies can be "on-the-fly" detected by the checker. The other dependencies need an extra computation time to be rebuilt (detection delay). This delay is influenced by the adopted selection strategy. In the paper, several selection strategies are proposed and evaluated with respect to the detection delay experienced by the checker.

1 Introduction

Many dependability problems, such as consistent global state selection [2, 9], detection of obsolete data [11], replicated data management [13] and distributed debugging [5], require the capability to detect *the concurrency or the causal dependency* of events forming a distributed computation, concurrently with the execution of the computation. A well-known approach to answer the question "*given a pair of events e and e', does e causally precede e'* [1]*?*" is (i) to use a *timestamping protocol* to timestamp the events of the distributed computation and (ii) to place a checker process that receives the timestamps of the events and provides the answer [7].

If the set of these timestamps can be structured as a partial order isomorphic to the one of the computation, we say that the protocol *characterizes causality*. Moreover, if the checker can detect a causality relation between two events e and e' as soon as it receives their timestamps we say that *a protocol characterizes causality on-the-fly*. A protocol characterizes causality *not on-the-fly* when the checker needs to receive the timestamps from other events (distinct from e and e') to detect the causality relation between them. This additional overhead at the checker produces, as most visible effect, a delay in perceiving a causality relation (detection delay).

A system of Vector Clocks (VC) [4, 12], also called Transitive Dependency Vectors, is an example of a timestamping protocol characterizing causality on-the-fly. More precisely, the vector clock local at a process P_i is a vector of n integers (one entry per process) such that the entry j counts the number of relevant events produced by the process P_j which causally precede the current event produced by P_i. To adhere to their semantic, during the computation, each application message carries a vector clock as a control information. The timestamp of an event produced by a process is the current value of the vector clock of the process. As proved in [3], unfortunately, vector clocks are not scalable as their size is equal to the number of processes. As, nowadays, local memory is cheap and available on-board in large quantity, the scalability issue does not concern vectors stored in local memory (at least if n is in a reasonable range). In a distributed setting, scalability is critical with respect to the control information attached to messages. Having a bounded, possibly fixed, control information size, would help protocol designers to develop efficient protocols for tracking causal dependencies.

In the paper we present *k-Dependency Vectors*, a scalable protocol for causality-tracking which associates each event e with a dependency vector $k\text{-}DV$ of size n. To this aim each process P_i is equipped with a vector of size n, $k\text{-}DV_i$, and each time P_i sends a message, the component $k\text{-}DV_i[i]$ plus $k-1$ ($1 \leq k \leq n$) distinct entries of the local dependency vector $k\text{-}DV_i$, selected according to some strategy, are piggybacked on the message as a control information. It is interesting to remark that when $k = n$,

*This work is partially supported by a grant of the EU in the context of the IST project "EU-PUBLI.com".

[1] This question actually corresponds to the detection of the *happened-before* relation [10] between two events of a computation.

Proceedings of the Eleventh Euromicro Conference on Parallel,Distributed and Network-Based Processing (Euro-PDP'03)
0-7695-1875-3/03 $17.00 © 2003 IEEE

k-dependency vectors become vector clocks. When $k = 1$, k-dependency vectors boil down to a timestamping protocol, namely Direct Dependencies, proposed by Fowler and Zwaenepoel in [5] in the context of distributed debugging. Thus this scalable protocol exploits a tradeoff between the size k of the piggybacked information and the detection delay (DTD). More precisely, DTD corresponds to the interval of time elapsed between the time the checker receives the information related to a pair of events e and e' and the time it is able to give the correct answer about the causality relation between e and e'.

We present several strategies to select the $k-1$ entries of local k-DVs. These strategies have a strong impact on DTD. We point out that when using a specific strategy, namely *the Fixed-Set* strategy, where each process piggybacks the *same* $k-1$ entries on each message (the Fixed-Set), all the causality relations $e \rightarrow e'$ such that e has been produced by one of the $k-1$ processes in the Fixed-Set, are on-the-fly detectable by a checker, i.e. DTD is zero as for vector clocks. If no constraint is imposed on the causality relations to be detected, we show that when each process P_i piggybacks on messages the $k-1$ entries of the local k-dependency vector related to the *last* $k-1$ distinct processes from which P_i received a message (Most-Recently-Received strategy), DTD is very small even for small values of k. Simulations show that 2-dependency vectors yield a reduction of 90% of the average value of DTD with respect to 1-dependency vectors (i.e. direct dependencies). On the contrary, if each process selects the $k-1$ entries *randomly*, then DTD is very close to the one of 1-dependency vectors.

The remaining of this paper is structured in 5 sections. Section 2 introduces the computational model. Section 3 presents a framework for timestamping protocols. Examples of causality-tracking protocols are introduced in Section 4. Section 5 introduces k-Dependency Vectors and surveys methods used in the literature to obtain a system of bounded dependency vectors. Section 6 concludes the paper.

2 Computation Model

A distributed computation consists of a finite set of n sequential application processes $\{P_1, \ldots, P_n\}$ without a common memory and communicating solely by exchanging messages. Each ordered pair of processes is connected by a reliable directed logical channel. Transmission delays are unpredictable but finite. The execution of each process P_i produces a totally ordered set of events E_i. An event may be either *internal* or it may involve communication (*send* or *receive* event). We denote by E the set of all the events of the computation, that is $E = \cup_{i=1}^{n} E_i$.

Following Lamport [10], we say that an event e *locally precedes* e' ($e \prec_l e'$) if e precedes e' on the same process. Moreover, each message m can be associated with its send and receive events denoted by $send(m)$ and $receive(m)$, respectively. Such a pair of events can be ordered by a relation of *message precedence* $\prec_m$. The causality ordering of events is based on Lamport's relation called *happened-before* or *causality relation*, and denoted by $\rightarrow$, defined as the transitive closure of the relation $\prec_l \cup \prec_m$. It is well known that $(E, \rightarrow)$ is a partial order of events. Two events e and e' are concurrent, denoted $e||e'$, if $\neg(e \rightarrow e')$ and $\neg(e' \rightarrow e)$.

The *causal past* $\mathcal{P}(e)$ of an event e is defined as the set $\mathcal{P}(e) = \{e' \in E \mid e' \rightarrow e\} \cup \{e\}$.

Moreover, given two distinct events e and e' with $e \rightarrow e'$, we say that a sequence of messages $< m_1, \ldots, m_h >$ is a *causal path* from e to e', denoted by $CP_e^{e'}$, if:

1. $e \prec_l send(m_1)$ or $e = send(m_1)$
2. $\forall i = 1, \ldots, h-1$ $receive(m_i) \prec_l send(m_{i+1})$
3. $receive(m_h) \prec_l e'$ or $e' = receive(m_h)$

We note that more than one causal path may exist between two distinct events e and e' such that $e \rightarrow e'$.

3 Causality-Tracking between Events

3.1 Timestamping protocols

To design efficient distributed algorithms one needs to track causal dependencies during the computation. Methods used to track these dependencies are based on timestamps of events produced by the execution of a timestamping protocol, which assign on-the-fly, that is during the evolution of the computation, to each event e, a value $\phi(e)$ (called timestamp of e) of a suitable domain $(D, <)$ whose relation $< \subseteq D \times D$ is antisymmetric. A *timestamping protocol* is usually characterized by:

(i) a *timestamping function* ϕ which establishes a correspondence between the events of a computation and the timestamps in D, and

(ii) a set of rules implementing the protocol which decide the control information piggybacked by messages used to update the timestamps.

We will denote as $D(E)$ the set of the timestamps assigned by the timestamping protocol to the events, i.e. $D(E) = \{\phi(e) \mid e \in E\}$.

The aim is to assign values in D to the events so that the transitive closure of $(D(E), <)$ of the timestamps assigned to the events may be an isomorphic embedding of $(E, \rightarrow)$. This can be formalized by requiring that the function ϕ *characterizes causality*, i.e., ϕ is injective and

$$\forall e, e' \in E,\ \phi(e) <^{+} \phi(e') \Leftrightarrow e \rightarrow e', \qquad (1)$$

being $<^+$ a partial order, that is the transitive closure of the relation $<$ on the set $D(E)$.

The Checker Process. We consider an additional observer process (i.e., the checker), whose role is to detect if a couple of events is causally related or concurrent only analyzing timestamps. Namely, each time an event e is generated by a process P_i, the checker receives a pair $(e, \phi(e))$ being $\phi(e)$ the timestamp associated with e. Its role is to detect the causal dependency or the concurrency between events just by analyzing the received timestamps. The checker can correctly detect dependencies only if the protocol characterizes causality.

3.2 Detection Delay

To answer the question "*given a pair of events e and e', does e causally precede e'?*", the checker needs to receive the timestamps associated with the events belonging to $\mathcal{P}(e')$. From an operational point of view, as the network can reorder messages, it is not sure that all these timestamps will be received by the checker before the receipt of the messages related to e'. If the checker requires a timestamp which has not yet been received, it waits until it has received the required information. This waiting actually creates a variable delay, namely *DeTection Delay* (DTD), defined as the time elapsed between the receipt of $(e, \phi(e))$ and $(e', \phi(e'))$ and the answer returned by the checker. The detection delay is influenced only by the late arrival at the checker of some events that belong either to the causal past of e or to the one of e'.

On-the-fly Causality-Tracking. We say that the timestamping function ϕ *characterizes causality on-the-fly* if ϕ characterizes causality, and the suborder of timestamps $(D(E), <)$ is already transitively closed (i.e., $<^+ \equiv <$). When the protocol characterizes causality on-the-fly, the checker can *"on-the-fly"* detect the causal dependency or the concurrency between events only comparing their timestamps, i.e. $DTD = 0$, as the structure of causality is represented in an isomorphic way by the suborder of timestamps $(D(E), <)$. On the contrary, let us remark that when property (1) is satisfied, but the suborder $(D(E), <)$ is not transitively closed, ϕ characterizes causality, but *not on-the-fly*, i.e., the checker might experience a delay in detecting a causality relation.

4 Causality-Tracking Protocols

Vector Clocks. In a system of vector clocks, introduced independently by Fidge and Mattern [4, 12], each event e is associated with a timestamp $\phi(e)$, denoted $e.VC$ and called vector clock, which is a vector of n integers (where n is the number of processes of the computation) such that $e.VC[i] = \mathcal{P}(e) \cap E_i$, that is $e.VC[i] = x$ when the x-th event of process P_i causally precedes e. During the computation, each application message carries a vector clock as control information.

Fidge and Mattern showed that this protocol characterizes causality on-the-fly ($DTD = 0$), that is given two vectors $e.VC$ and $e'.VC$ associated with the events e and e', respectively:

$$e \rightarrow e' \Leftrightarrow e.VC < e'.VC \tag{2}$$

where $e.VC < e'.VC$ iff $\forall i = 1, \ldots, n, \quad e.VC[i] \leq e'.VC[i] \wedge e.VC \neq e'.VC$.
Moreover, $e||e'$ iff $\neg(e.VC < e'.VC) \wedge \neg(e'.VC < e.VC)$.

From an operational point of view the protocol works as follow: each process P_i endows a vector VC_i of n integers, where $VC_i[j]$ represents the index of the most recent event of process P_j known by P_i. Then, the value of VC_i represents actually the timestamp of the event currently produced by P_i. During the computation, each application message carries the whole local vector clock as control information.

The vector VC_i is updated according to following rules:

1. when P_i starts its execution, each component of VC_i is initialized to zero;
2. when an event is generated on process P_i: $VC_i[i] = VC_i[i] + 1$;
3. when a message m is sent by P_i, a copy of VC_i (VC_m) is piggybacked on m;
4. when a message m sent by process P_j is received by P_i: $\forall h = 1, \ldots, n, \quad VC_i[h] = max(VC_i[h], VC_m[h])$.

The major drawback of vector clocks lies in the fact that they have poor scalability. In fact, its implementation requires an entry for each one of the n processes in the computation. In [3] it is proved that given a distributed computation with n processes, there is always a possible combination of events occurring in the computation whose causality can only be detected on-the-fly by vector clocks with n entries.

Answers to scalability issue. As far as we know, two approaches have been proposed to tackle the problem of the size of piggybacked information. Each one relies on a specific tradeoff:

- *Trading* piggybacked information (I_p) *for* local memory overhead (I_l). This class includes "efficient implementations of vector clocks" presented in [8, 15]. These implementations try to move locally as much as

possible the complexity of managing a vector clocks system while maintaining the on-the-fly characterization of causality;

- *Trading I_p* and *I_l for* missing some concurrencies between events (i.e., concurrent events can be perceived as ordered). This tradeoff is exploited by plausible clocks [17]. This system of clocks actually bounds the size of I_p and I_l to some integer k less than n. As a consequence plausible clocks does not characterize causality, i.e., the property (1) is not necessarily satisfied;
- *Trading I_p for* a delay in detecting a causality relation. This tradeoff is captured by the protocol proposed in this paper and it has Direct Dependencies protocol [5] as a particular case when considering k equal to one. k-dependency vectors actually bounds the size of I_p to some integer k less than n. They characterize causality but not on-the-fly.

Efficient implementations of vector clocks. In [8, 15], it has been proposed efficient implementations of vector clocks that address the *reduction of the size of control information*, i.e., they do their best to piggyback timestamps whose size is less than n as control information. These are based on this idea: when P_i sends a message to P_j, it may piggyback only the entries that have been modified since its last sending to this process P_j.

A system of vector clocks adopting this technique does not add extra delay to detect the concurrency or the causality relation between two events. These improvements are expected to save communication bandwidth at the cost of a local memory overhead. On the other hand, the amount of control information is variable and, in the worst case, it can grow up to n.

Plausible vector clocks. Torres and Ahamad [17] proposed a simple mechanism, called *Plausible Vector Clocks* (sometimes also called approximate vector clocks), that allows to manage vectors of k integers, with $k < n$. The implementation is similar to the vector clocks one, but each process locally maintains a vector of size k and piggybacks this vector upon messages. In this case two events could be perceived as dependent while they are actually concurrent. More formally, if $e.PC$ and $e'.PC$ are the two timestamps associated with two events e and e', then

$$e \rightarrow e' \Rightarrow e.PC < e'.PC.$$

This method can be used in some applications in which one could be interested in capturing the causality between events not their concurrency. As plausible clocks may violate concurrency, they are useful in systems where ordering of concurrent events impacts performance not correctness. When $k \neq n$, the protocol does not characterizes causality, as the transitive closure of timestamps suborder is an extension (not an isomorphic embedding) of the computation $(E, \rightarrow)$. When $k = 1$ the resulting protocol is the Lamport's scalar clock system [10], while when $k = n$ we get classical vector clocks.

5 k-Dependency Vectors

In this section we introduce k-dependency vectors which is a general scheme for causality-tracking exploiting the tradeoff between the size of the control information piggybacked on messages and the detection delay. Next, we present two interesting properties on these vectors. This section also presents effective strategies that reduce the detection delay at a checker for those causality relations which are not on-the-fly detectable. The section finally presents a simulation study comparing different dependency vectors systems with respect to the detection delays.

Basic idea. The idea behind k-dependency vector timestamping protocol is the following: each time a process sends a message m, it attaches to the message k entries of the local dependency vector selected as follows (for each set of events X, let X_i be the set $X \cap E_i$):

- a coding of $\mathcal{P}(send(m))_{sender(m)}$;
- $k-1$ ($k < n$) codings of distinct sets $\overline{\mathcal{P}}(send(m))_{\ell_1}, \ldots \overline{\mathcal{P}}(send(m))_{\ell_{k-1}}$ (with $\ell_j \neq i$) selected according to some strategy (see Section 5.2), where each $\overline{\mathcal{P}}(send(m))_{l_i}$ *is the largest prefix-closed subset of* $\mathcal{P}(send(m))_{l_i}$ known to the sender (i.e., $\overline{\mathcal{P}}(send(m))_{l_i} \subseteq \mathcal{P}(send(m))_{l_i}$).

In the following we denote as $\overline{\mathcal{P}}(e)|_m$ (with $\overline{\mathcal{P}}(e)|_m \subseteq \overline{\mathcal{P}}(e) \subseteq \mathcal{P}(e)$) the set of events $\mathcal{P}(send(m))_{sender(m)} \cup \overline{\mathcal{P}}(send(m))_{\ell_1} \cup \ldots \cup \overline{\mathcal{P}}(send(m))_{\ell_{k-1}}$ whose codings are piggybacked by a process onto m just before sending it.

As $k < n$, the sender transfers a coding of a part of what it knows about its causal past. As a consequence, let e be an event, the set $\cup_{h=1,\ldots n}\overline{\mathcal{P}}(e)_h$, denoted $\overline{\mathcal{P}}(e)$, is a subset of $\mathcal{P}(e)$ as depicted in Figure 1. If $k = n$, we obtain a system of vector clocks and $\overline{\mathcal{P}}(e)|_m = \overline{\mathcal{P}}(e) = \mathcal{P}(e)$.

The protocol From an operational point of view, each process P_i maintains a k-dependency vector of n integers, $k\text{-}DV_i$, which represents the timestamp of the event e currently produced by P_i ($e.k\text{-}DV$). Each time a process P_i sends a message m to process P_j, it selects according to some policy $k-1$ entries $\ell_1, \ldots \ell_{k-1}$ from its local dependency vector plus the i-th entry, and then it piggybacks only this information, which corresponds to the coding of

$\overline{\mathcal{P}}(e)|_m$, onto m. k-DV_i is updated according to the following rules (rules 1 and 2 are omitted as they are similar to the ones of VC):

3″. when a message m is sent by P_i to P_h, a set I_m of k pairs $(process\_index, k\text{-}DV_i[process\_index])$ is piggybacked on m. I_m is composed of:

- the pair $(i, k\text{-}DV_i[i])$;
- $k-1$ pairs $(\ell_j, k\text{-}DV_i[\ell_j])$, $j = 1, 2, \ldots, k-1$ *and* $\ell_j \neq i$, where $\ell_1, \ldots, \ell_{k-1}$ are distinct indices selected according to some policy.

4″. when a message m is received by P_i:

$$\forall (\ell, h) \in I_m, \quad k\text{-}DV_i[\ell] = max(h, k\text{-}DV_i[\ell])$$

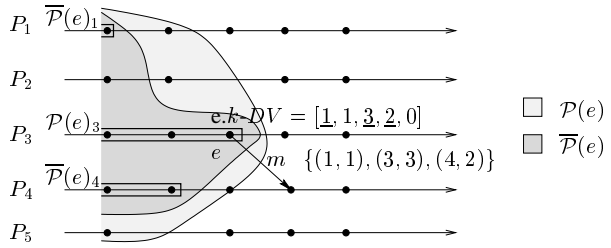

Figure 1. Example of k-Dependency Vectors

As an example, Figure 1 shows a computation using 3-dependency vectors. When process P_3 sends message m (event e), it piggybacks on m the pair $(3, DV_3[3])$ plus two selected pairs which correspond, in this case, to the ones relative to P_1 and to P_4 (the selected entries are underlined in the figure). In [1, 14] it has been shown that the checker is able to reconstruct the vector clocks associated with e and e' by recursively exploring direct dependencies of the events in the causal past of e and e'. As k-dependency vectors always piggyback the direct dependency, they can reconstruct the vector clock associated with each event as the direct dependencies protocol.

5.1 Properties

As k-dependency does not transfer all the causal past of the sending event to the receiver, there can be cases in which a causality relation $e \rightarrow e'$ cannot be detected on-the-fly by the checker. The following property states the condition in which a checker can detect the correct causality relation between two events e and e' as soon as it receives their timestamps.

Property 1 *Let e be an event produced by P_i. We have:*

$$e.k\text{-}DV[i] \leq e'.k\text{-}DV[i] \Rightarrow e \rightarrow e'$$

Proof. If $e.k$-DV$[i] \leq e'.k$-DV$[i]$, then the $e.k$-DV$[i]$-th event of P_i, that is e, locally precedes or coincides with the $e'.k$-DV$[i]$-th event of P_i. Since this one causally precedes e', one has $e \rightarrow e'$. □

There can be cases in which $e \rightarrow e'$ and $e.DV[i] > e'.DV[i]$. This implies that several events can be "on-the-fly" perceived as concurrent by a checker, when they are actually causally ordered[2]. The following property states the condition in which a causality relation is detected on-the-fly by a checker.

Property 2 *Let e and e' be two events of a distributed computation $(E, \rightarrow)$ which adopts the k-dependency vectors protocol to timestamp its events. A checker process is able to detect on-the-fly $e \rightarrow e'$ iff a causal path $CP_e^{e'} =< m_1, \ldots, m_h >$ exists such that:*

$$e \in \bigcap_{i=1}^{h} \overline{\mathcal{P}}(send(m_i))|_{m_i}. \tag{3}$$

Proof. Let e be an event produced by process P_i and $CP_e^{e'} =< m_1, \ldots, m_h >$ be a causal path from e to e' that satisfies the property (3), i.e., each $send(m_j)$, for $j = 1, \ldots, h$, piggybacks the coding of $\overline{\mathcal{P}}(send(m_j))_i$. By Property 1, to prove that a checker process can detect on-the-fly $e \rightarrow e'$, it is sufficient to prove that $e.k$-DV$[i] \leq e'.k$-DV$[i]$. This thesis is a direct consequence of following considerations:

- $e.k$-DV$[i] \leq send(m_1).k$-DV$[i]$, as $e \prec_l send(m_1)$ or $e = send(m_1)$ (by definition of causal path);
- $send(m_j).k$-DV$[i] \leq receive(m_j).k$-DV$[i]$ $(i = 1, \ldots, h)$ (by rule 4" of k-dependency vectors protocol);
- $receive(m_j).k$-DV$[i] \leq send(m_{j+1}).k$-DV$[i]$ $(i = 1, \ldots, h-1)$, as $receive(m_j) \prec_l send(m_{j+1})$ (by definition of causal path);
- $receive(m_h).k$-DV$[i] \leq e'.k$-DV$[i]$, as $receive(m_h) \prec_l e'$ or $e' = receive(m_h)$ (by definition of causal path).

Let us now suppose that a checker process is able to detect $e \rightarrow e'$ as soon as it receives their timestamps. Clearly, if there were no causal path from e to e' satisfying property (3), it would be $e.k$-DV$[i] > e'.k$-DV$[i]$. In this case, the checker would not be able to detect on-the-fly the causal dependency. So, we can conclude that such a causal path must exist. □

As an example, in Figure 2 the checker is able to detect on the fly the relation $e_1 \rightarrow e'$, but it is not able to detect the

[2]Let us remark that when considering vector clocks Property 1 keeps also in the other direction.

$e_2 \rightarrow e'$ one. In fact, in the first case there is the causal path $CP_{e_1}^{e'} =< m_1, m_2, m_3 >$ such that $e_1 \in \bigcap_{i=1}^{3} \overline{\mathcal{P}}(e_1)|_{m_i}$, while in the other one $e_2 \notin \overline{\mathcal{P}}(e_3)|_{m_3}$.

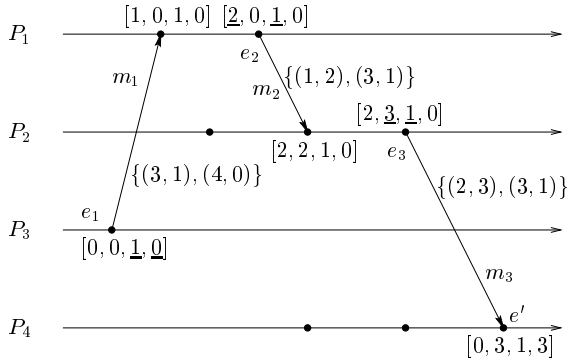

Figure 2. A distributed computation using 2-dependency vectors

For all the couples of events e and e' that either (i) are concurrent or (ii) are causally dependent (i.e. $e \rightarrow e'$) but do not satisfy Property 2, a checker needs to reconstruct, at least partially, the vector clocks of e and e' and then compare them.

5.2 How to Select $k-1$ Entries of a Dependency Vectors.

Property 2 defines a set of causality relations between pairs of events that can be detected by a checker just by comparing the k-dependency vectors of the events. To detect the other causality relations a checker needs to rebuild, at least partially, the vector clocks related to the involved events. This may introduce a detection delay at the checker as remarked in the previous section.

Here we consider the following problem: "how to select $k-1$ entries of a dependency vectors in order to minimize the detection delay at the checker?".

Simple selection heuristics could be: the *random* and the *static* one. The random strategy selects the $k-1$ entries randomly. In the static strategy each process selects the *same* $k-1$ entries each time it sends a message. However, despite their simplicity, both these heuristics do not attack the problem of the minimization of the detection delay. At this aim we introduce the following two strategies, namely, the *Most Recently Received* (MRR) strategy and the *Fixed-Set* strategy.

MRR Strategy. From previous Section it can be argued that the detection delay of a causality relation not on-the-fly detectable by a checker occurs as a consequence of some *causal ordering violation* at the checker. From the point of view of a process P_i at the time it is sending a message (event e), the best heuristic to minimize the detection delay is to make on-the-fly detectable by a checker the causality relations that involve events that have more probability to create a causality ordering violation at the checker with e. If a message m has been received by P_i a lot of time before e, it will be extremely unlikely that the information related to the event $send(m)$ will cause a causality violation at the checker with the information related to e, i.e. $checker(e) \rightarrow_l checker(send(m))$ (see Figure 3).

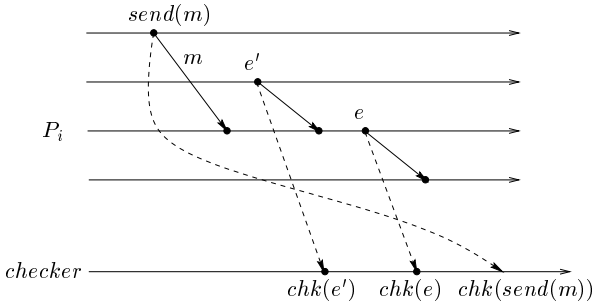

Figure 3. Example of a low probability causality ordering violation

Hence the events more risky of causality ordering violations at the checker are the ones connected to the last receipt of process P_i before producing e (such as event e' in Figure 3). An effective heuristic consists, therefore, in selecting the $k-1$ entries of a dependency vector of a process P_i related to distinct senders of the $k-1$ Most Recently Received (MRR strategy) messages by P_i[3].

In this way the effect is twofold. On one side we make on-the-fly detectable by a checker the causality relations that involve events that could create with high probability a causality ordering violation with e (for example $e' \rightarrow e$ in Figure 3). On the other side, when considering not on-the-fly detectable causality relations, it will be extremely likely that all the information related to dependency vectors necessary as an input to the reconstruction algorithm of the previous section will be present at the time it runs (i.e. no delay is introduced to wait for some dependency vector to arrive).

Fixed-Set Strategy. Let us consider a class of applications where one is interested in capturing only a subset of all causality relations. This subset contains all the causality relations $e \rightarrow e'$ such that e is an event produced by a process in the set $P_{\ell_1}, \ldots P_{\ell_{k-1}}$ ($k < n$). These processes form the Fixed-Set.

[3] If such processes are less than $k-1$, one option is to fill the missing pairs with processes having a non-zero entry in $k\text{-}DV$.

The Fixed-Set strategy works as follows: each process P_i piggybacks on each message m its entry $k\text{-}DV_i[i]$ plus the entries related to processes in the Fixed-Set.

If $e \rightarrow e'$ and e has been produced by a process in the Fixed-Set, then this relation is on-the-fly detectable by a checker as the entry of the process producing e, say P_{ℓ_i}, is piggybacked onto *all* messages of the computation. As a consequence, e satisfies Property 2. Hence for this subset of causality relations, k-dependencies are equivalent to vector clocks.

Note that, using this heuristic, a process does not need to piggyback couples (process identifiers, entry of the local dependency vector) on each message as all processes agree on the members of the Fixed-Set. Then information on the process identifier can be omitted.

5.3 Simulation study

The simulation study carries out a performance comparison between k-dependency vectors, using MRR and random, and direct dependency vectors (i.e. 1-Dependency vectors). In particular, given a pair of events e and e', we measure the *DeTection Delay* (DTD) as a function of the number of processes of the computation. We simulated a point-to-point environment in which each process can send messages to any other process and the destination of each message is an uniformly distributed random variable. Each process generates an internal, send or receive event with the same probability[4]. Transmission delays of each directed point-to-point channel are distinct uniformly distributed random variables. These distributions are distinct to maximize the probability of causality ordering violations (as the one depicted in Figure 3). Each simulation consists of one million of events and for each value of n in the set $\{5, 7, 10, 15, 20, 25, 35, 50, 70, 100\}$ we plot the ratio (R) between the mean DTD of k-DV (adopting MRR and the random strategies with three distinct value of k) and the mean DTD of 1-DV. We did ten runs of each simulation with different seeds and the results were within 6% of each other. Thus variance is not reported in the plots. Results of the simulation study are plotted in Figure 4 and Figure 5.

In Figure 4, 2-DV *adopting MRR* shows a reduction in DTD of 10 times with respect to 1-DV almost independent of the number of processes. This saturation behavior can be justified by pointing out that $k\text{-}DV$ using MRR, independently of the number of processes, always chooses to transfer information related to events with the highest probability to create a causality ordering violation at the checker, that is to cause a delay. Moreover, an additional increment of k over a certain value produces a little reduction of R as k-DV transmits information related to events that cause a causality ordering violation with small probability. The

[4]Receive events are generated in response to generated send events.

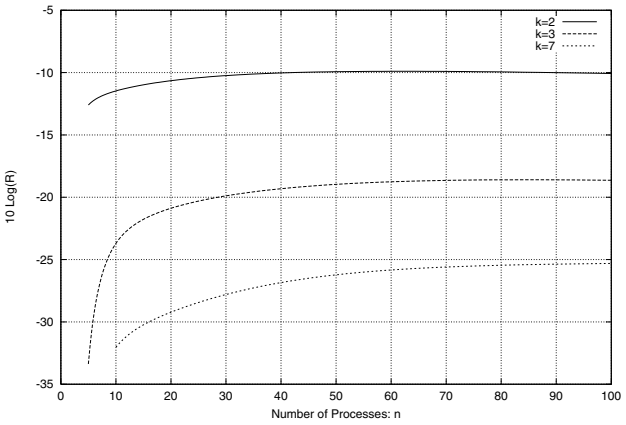

Figure 4. R *vs* Number of processes n

graphs show that a system of k-dependency vectors using MRR strategy is very scalable, being k almost uniquely dependent on the desired DTD improvement over 1-DV. Moreover, they show that there is no need to give k large values, since $k = 7$ already gives a reduction of over 300 times in a system of 100 processes.

In Figure 5, we finally compare 1-DV with k-DV when adopting two distinct selection policies, namely MRR, and the random one. The graphs of Figure 5 give an idea on (i) how a selection strategy impacts the detection delay and (ii) the real performance distance between MRR and the random strategy. It is interesting to remark that, all plots related to the random strategy are very close each other and for a number of processes larger than 30, their performance are similar to the one of 1-DV independently of the value of k. On the other hand, when adopting MRR as the selection strategy the value of k influences the performance as shown in Figures 4 and 5.

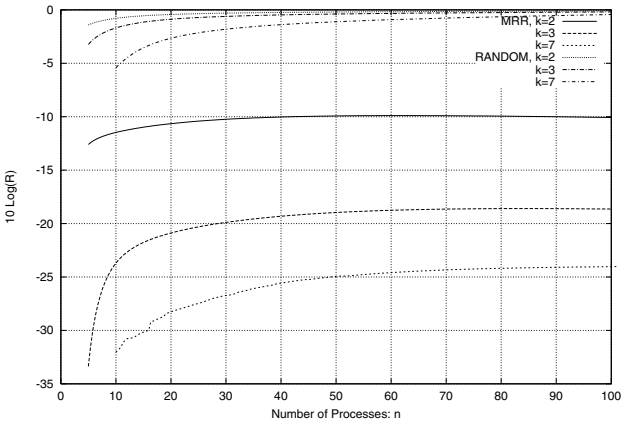

Figure 5. R *vs* Number of processes n

6 Conclusions

Up to now the only known ways to get bounded dependency vectors were missing some concurrency between events (i.e., accepting some concurrent events appear as causally ordered) or having a local storage overhead. This paper pointed out another tradeoff: bounded dependency vectors can be traded for missing some on-the-fly detection of causality relation at a checker. We then presented a general scheme for causality-tracking, namely k-dependency, which lies on that tradeoff. This scheme has direct dependencies and vector clocks as extreme cases when considering k equal to one and to n, respectively. This scheme provides scalability, with respect to control information piggybacked on application messages, by attaching only $k \leq n$ entries of a vector of integers on each message. These entries are selected from a vector of n entries according to some strategy. Simulations showed that when k-dependency vectors adopt the MRR strategy even for small (and fixed) value of k, the delay in detecting a causality relation by a checker is kept small. Let us finally remark that for particular classes of distributed applications where one is interested in detecting causality relations $e \rightarrow e'$ such that e has been produced by a process in a set of $k-1$ processes, k-dependency vectors adopting the Fixed-Set strategy makes all such causality relations on-the-fly detectable by a checker.

References

[1] R. Baldoni, G. Cioffi, J.M. Hélary, and M. Raynal, "Direct Dependency-Based Determination of Consistent Global Checkpoints", to appear in *Journal of Computer Systems Science and Engineering*, 2001.

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

[3] B. Charron-Bost, "Concerning the size of logical clocks in distributed systems", *Information Processing Letters*, 39:11-16, 1991.

[4] G. Fidge, "Timestamps in message passing system that preserve the partial ordering", *Proc. 11th Australian Computer Science Conf.*, pp. 55-66, 1988.

[5] J. Fowler and W. Zwaenepoel, "causality distributed breakpoints", *Proc. of 10th IEEE International Conf. on Distributed Computing Systems*, pp. 134-141, 1990.

[6] E. Fromentin, C. Jard, G.V. Jourdan, and M. Raynal, "On-The-Fly Analysis of Distributed Computations", *Information Processing Letters*, 54(5):267-274, 1995.

[7] V. K. Garg, *Principles of Distributed Systems*, Kluwer Academic Press, 274 pages, 1996.

[8] J.M. Hélary, G. Melideo and M. Raynal, "Tracking Causality in Distributed Systems: a Suite of Efficient Protocols", *Proc. 7th Int. Colloquium on Structural Information and Communication Complexity (SIROCCO)*, Carleton University Press, pp. 181-195, L'Aquila (Italy), June 2000.

[9] J.M. Hélary, R. Netzer and M. Raynal, "Consistency issues in distributed checkpoints", *IEEE Transactions on Software Engineering*, 25(2):274-281, 1999.

[10] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", *Comm. ACM*, 21(7):558-564, 1978.

[11] B. Liskov and R. Ladin, "Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection", *Proc. 5th ACM Symposium on Principles of Distributed Computing*, pp. 29-39, 1986.

[12] F. Mattern, "Virtual time and global states of distributed systems", *Parallel and Distributed Algorithms*, M. Cosnard and P. Quinton eds., PP. 215-226, 1988.

[13] D.S. Parker *et al.*, "Detection of Mutual Inconsistency in Distributed Systems", *IEEE Transactions on Software Engineering*, SE9(3):240-246, 1983.

[14] R. Schwarz and F. Mattern, "Detecting causality relations in distributed computations: in search of the holy grail", *Distributed Computing*, 7(3):149-174, 1994.

[15] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks", *Information Processing Letters*, 43(1):47-52, 1992.

[16] R.E. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems", *ACM Transactions on Computer Systems*, 3(3):204-226, 1985.

[17] F.J. Torres-Rojas and M. Ahamad, "Plausible Clocks: constant size logical clocks for distributed systems", *Proceedings of the International Workshop on Distributed Algorithms*, pp. 71-88, 1996.