Distrib. Comput. (1999) 12: 179–195

DISTRIBUTED COMPUTING

Plausible clocks: constant size logical clocks for distributed systems*

Francisco J. Torres-Rojas, Mustaque Ahamad

College of Computing, Georgia Institute of Technology, USA (e-mail: {torres, mustaq}@cc.gatech.edu)

Received: January 1997 / Accepted: January 1999

Summary. In a Distributed System with N sites, the precise detection of causal relationships between events can only be done with vector clocks of size N. This gives rise to scalability and efficiency problems for logical clocks that can be used to order events accurately. In this paper we propose a class of logical clocks called plausible clocks that can be implemented with a number of components not affected by the size of the system and yet they provide good ordering accuracy. We develop rules to combine plausible clocks to produce more accurate clocks. Several examples of plausible clocks and their combination are presented. Using a simulation model, we evaluate the performance of these clocks. We also present examples of applications where constant size clocks can be used.

Key words: Logical clocks – Causality detection – Distributed algorithms

1 Introduction

In large scale distributed systems, efficient access to shared information requires the use of caching and replication. In such an environment, it is necessary to order read and update operations on an object to determine its most recent value. Logical clocks have been explored for ordering events in distributed systems. These clocks do not require synchronized physical clocks and can be implemented by including additional information with messages exchanged in the system. Although vector clocks, one example of logical clocks, can precisely order events of a distributed system and detect concurrent events, they are expensive to maintain and manipulate since vectors of integers must be included in messages and it is necessary to compare vector times to determine the order between operations. Besides, since vector clocks have a component for each site in the system, they are not easily scalable.

Scalar logical clocks can be implemented efficiently (e.g., Lamport Clocks), but when events are timestamped with these clocks, two events may appear to be ordered according to their timestamps even when they are concurrent. In a replicated object system, this could lead to unnecessary consistency operations. For example, in the causal consistency implementation described by Ahamad, John, et al. [2, 16], if a new object value written by operation **o** is fetched at a site, existing object copies at the site are removed from its cache if the operations that produced them causally precede **o**. This is done because the cached object copies may potentially have been overwritten by more recent operations that were executed before **o**. If scalar clocks are used to determine the causal orderings between operations, an object produced by an operation which is concurrent with **o** may unnecessarily be removed from the site cache. Such unnecessary removals and extra communications needed when such objects are accessed in the future can be avoided if more discerning clocks are used to determine the ordering between operations.

In this paper, we explore logical clocks that make use of multiple components, as in vector clocks, to provide a high level of ordering accuracy. However, they can be implemented in a scalable fashion because the number of components in them is independent of the number of sites in the distributed system. Such clocks are useful in systems where placing an ordering on concurrent events only impacts performance and not correctness. This is true for many consistency maintenance schemes and resource allocation algorithms. Thus, we explore efficiently implemented clocks that may order a small number of concurrent events but do not significantly affect the performance of algorithms that use such clocks due to their high level of ordering accuracy. We call these *plausible* clocks. We describe several such clock systems and present rules to combine them to produce more accurate clocks. We study the performance of these clocks using a simulation model for two different distributed systems. These simulations show that plausible clocks can provide high ordering accuracy in many systems. For example, the experiments show that in a group of client/server systems with 76 sites (1 server, 75 clients) that include 96 million event pairs, a plausible clock with just 7 components ordered, on the average, 93% of the event pairs in the same way as a vector clock which requires 76 components.

* This work was supported in part by NSF grant CDA-9501637 and CCR-9619371.

This paper is a significant revision of a previous version that appeared in the Proceedings of the Workshop on Distributed Algorithms (WDAG), 1996 [27]. We have refined the model used to represent the clock systems and present additional details of the workings of plausible clocks. A new section, that illustrates where plausible clocks can be used, has been added. We also include a much expanded discussion of related work and its comparison with plausible clocks.

Section 2 presents some background material on logical clocks. We introduce the concept of plausible clocks in Sect. 3. Section 4 explains how such clocks can be combined and demonstrates that the combination rules preserve correctness and provide improved accuracy. Some examples of constant size clocks are described in Sect. 5. Section 6 reports the simulation model used to evaluate the proposed clocks together with the obtained results. We outline some applications of plausible clocks in Sect. 7. Related work is examined in Sect. 8. Finally, Sect. 9 offers the conclusions of this paper.

2 Logical clocks

This section reviews some basic material on logical clocks. We consider a system where a set of processes communicate exclusively by exchanging messages. There is neither common memory nor a common physical clock, and the relative speed of the processes is unknown. It is assumed that all communication is asynchronous and point-to-point, and that all messages are delivered correctly. Although failures are an important issue in large scale distributed systems, we do not consider failures in this work.

2.1 Lamport clocks

In 1978, Leslie Lamport proposed the concept of *logical clocks* to order events in distributed systems [18]. A logical clock consists of a mapping L from events to the set of integers that captures the causal order between events.

Definition 1. The *local history* of site i is a sequence of events $\boldsymbol{H}_i = \mathbf{e}_{i1}\mathbf{e}_{i2}\ldots$ that are executed on site i. There is a total order on the local events of each site. The *global history* $\boldsymbol{H}$ of the Distributed System is the set of all events occurring at all sites of the system. Lamport [18] defines the *causality relation* "$\longrightarrow$" over events as the smallest relation such that:

- If $\mathbf{e}_{i\mathrm{j}}$ and $\mathbf{e}_{i\mathrm{k}} \in \boldsymbol{H}_i$ and j < k, then $\mathbf{e}_{i\mathrm{j}} \longrightarrow \mathbf{e}_{i\mathrm{k}}$.
- If $\mathbf{e}_{i\mathrm{m}}$ is **send**(M), $\mathbf{e}_{j\mathrm{n}}$ is **receive**(M) and M is the same message in both cases, with arbitrary i, j, m and n, then $\mathbf{e}_{i\mathrm{m}} \longrightarrow \mathbf{e}_{j\mathrm{n}}$.
- $\forall\, \mathbf{a}, \mathbf{b}, \mathbf{c} \in \boldsymbol{H}$ if $\mathbf{a} \longrightarrow \mathbf{b}$ and $\mathbf{b} \longrightarrow \mathbf{c}$ then $\mathbf{a} \longrightarrow \mathbf{c}$.

If neither $\mathbf{a} \longrightarrow \mathbf{b}$ nor $\mathbf{b} \longrightarrow \mathbf{a}$ holds between two different events **a** and **b**, then **a** and **b** are *concurrent*. This situation is denoted as $\mathbf{a} \parallel \mathbf{b}$. Since the causality relation $\longrightarrow$ is irreflexive and transitive, it defines a strict partial order $\langle \boldsymbol{H}, \longrightarrow \rangle$ over the events of the system. □

To implement a Lamport Clock, each site i maintains an integer counter $\mathbf{L}_i$ that initially has a value of zero. Each message sent by site i includes a timestamp which is the value of $\mathbf{L}_i$ when the message was sent.

$\mathbf{L}_i$ is updated according to 3 rules:

- **L0)** Initial value:

  $\mathbf{L}_i = 0$

- **L1)** Before an event (other than a **receive**) is executed at site i:

  $\mathbf{L}_i = \mathbf{L}_i + 1$

- **L2)** When a message with timestamp **D** is received by site i:

  $\mathbf{L}_i = \max(\mathbf{L}_i, \mathbf{D})$
  $\mathbf{L}_i = \mathbf{L}_i + 1$

If **a** is an event of $\boldsymbol{H}_i$, then $L(\mathbf{a})$ is the value of $\mathbf{L}_i$ when **a** is executed[1]. Lamport clocks exhibit the *weak clock condition*, that states that $\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H}$:

$$\mathbf{a} \longrightarrow \mathbf{b} \Rightarrow L(\mathbf{a}) < L(\mathbf{b})$$

Lamport Clocks capture the order between causally related events but they do not detect concurrency between events and just by inspecting two timestamps, we are not able to decide if the associated events are causally related or concurrent (hence the name weak clock condition).

2.2 Vector clocks

Vector clocks were independently proposed by Fidge [10, 11, 12] and Mattern [19]. They consist of a mapping V from events to integer vectors. Each site i keeps an integer vector $\mathbf{V}_i$ of N entries, where N is the number of sites in the distributed system. Site i keeps its own logical clock in $\mathbf{V}_i[i]$, while $\mathbf{V}_i[j]$ represents the current knowledge that site i has of the activity at site j. Vector clocks are updated when events occur at the local site or when messages are received (all messages include the timestamp of their corresponding **send** event).

Site i updates $\mathbf{V}_i$ according to the rules:

- **V0)** Initial value:

  $0 \leq j \leq \mathrm{N} - 1 : \mathbf{V}_i[j] = 0$

- **V1)** Before an event (other than a **receive**) is executed at site i:

  $\mathbf{V}_i[i] = \mathbf{V}_i[i] + 1$

- **V2)** When a message with timestamp **W** is received by site i:

  $0 \leq j \leq \mathrm{N} - 1 : \mathbf{V}_i[j] = \max(\mathbf{V}_i[j], \mathbf{W}[j])$
  $\mathbf{V}_i[i] = \mathbf{V}_i[i] + 1$

If $\mathbf{a} \in \boldsymbol{H}_i$, then $V(\mathbf{a})$ is the value of $\mathbf{V}_i$ when **a** is executed.

Definition 2. Given two vector times **V** and **W**, the following tests are defined:

[1] For all the logical clocks presented in this paper, an event is timestamped *after* the local clock has been updated.

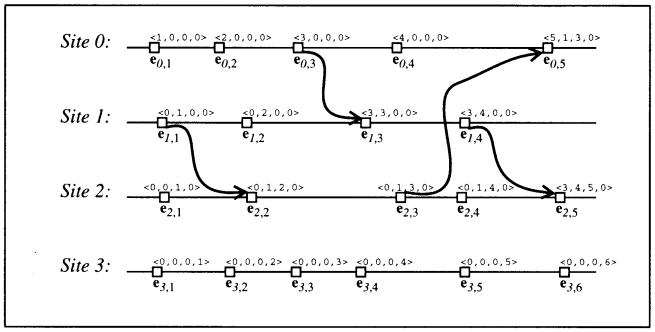

Fig. 1. Vector clock timestamps of events in an execution

$\mathbf{V} = \mathbf{W} \Leftrightarrow 0 \leq j \leq \mathrm{N}-1 : \mathbf{V}[j] = \mathbf{W}[j]$
$\mathbf{V} \leq \mathbf{W} \Leftrightarrow 0 \leq j \leq \mathrm{N}-1 : \mathbf{V}[j] \leq \mathbf{W}[j]$
$\mathbf{V} < \mathbf{W} \Leftrightarrow \mathbf{V} \leq \mathbf{W}$ and $\exists$ j such that $\mathbf{V}[j] < \mathbf{W}[j]$
$\mathbf{V} \parallel \mathbf{W} \Leftrightarrow \exists$ k such that $\mathbf{V}[k] < \mathbf{W}[k]$ and $\exists$ j such that $\mathbf{V}[j] > \mathbf{W}[j]$. □

Mattern [19] proved that there is an isomorphism between the times read from a vector clock when events are executed and the causality relation among events in $\boldsymbol{H}$, since $\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H}$:

$$\begin{aligned} \mathbf{a} = \mathbf{b} &\Leftrightarrow V(\mathbf{a}) = V(\mathbf{b}) \\ \mathbf{a} \longrightarrow \mathbf{b} &\Leftrightarrow V(\mathbf{a}) < V(\mathbf{b}) \\ \mathbf{a} \parallel \mathbf{b} &\Leftrightarrow V(\mathbf{a}) \parallel V(\mathbf{b}) \end{aligned} \quad (1)$$

Condition **(1)** is called the *strong clock condition*. A clock that satisfies this condition *characterizes* causality, while a clock that only satisfies the weak clock condition is *consistent with causality* [22, 25].

Figure 1 shows a simple execution of a distributed system with 4 sites. The arrows represent the sending and receiving of a message. Each event has been timestamped with its corresponding vector clock. It is easy to verify that the causal relations between any pair of events are correctly established by using the tests presented in Definition 2.

2.3 Disadvantages of vector clocks

Vector clocks precisely capture the ordering between events in a distributed system. However, they have the major disadvantage of not being constant in size: the implementation of vector clocks requires an entry for each one of the N sites in the system. If N is large, several problems arise. There are growing storage costs because each site must reserve space to keep its local version of the vector clock and, depending on the particular system, vector times associated with certain events must be stored as well. All messages of a distributed computation are tagged with timestamps read from vector clocks, which could add considerable overhead to communication in the system. Also, comparison of vector timestamps to determine ordering between events will have high processing overhead for large N. Thus, vector clocks have poor scalability.

Charron-Bost [6] proved that given a Distributed System with N sites, there is always a possible combination of events occurring in the system whose causality can only be captured by vector clocks with N entries. It may be possible to design a different mechanism to determine causality between events, however Charron-Bost's result [6] indicates that if such a mechanism characterizes the causality relation, it would have a size O(N). This discourages any attempt to define some kind of clock that, while constant in size, completely captures the causality relation.

3 Plausible clocks

In this section we propose a class of clocks that do not characterize causality completely, but are scalable because they can be implemented using constant size structures. Furthermore, under appropriate circumstances, they can provide a high level of ordering accuracy.

Definition 3. Let a *timestamp* be a structure that represents an instant in time as observed by some site. The particular details of this structure are left open. For a distributed system with global history $\boldsymbol{H}$, a *Time Stamping System* (TSS) $\boldsymbol{X}$ is a pair $(\langle \boldsymbol{S}, \xrightarrow{X} \rangle, \boldsymbol{X}.\mathbf{stamp})$, where:

$\boldsymbol{S}$ is a set of timestamps.
$\xrightarrow{X}$ is an irreflexive and transitive relation defined on the elements of $\boldsymbol{S}$.
$\langle \boldsymbol{S}, \xrightarrow{X} \rangle$ is a strict partial order.
– X.**stamp** is the *timestamping function* mapping $\boldsymbol{H}$ to $\boldsymbol{S}$.

For all timestamps v, $w \in \boldsymbol{S}$, we define the additional relations:

$v \overset{X}{=} w \Leftrightarrow v = w$
$v \xleftarrow{X} w \Leftrightarrow w \xrightarrow{X} v$
$v \overset{X}{\parallel} w \Leftrightarrow \neg(v \overset{X}{=} w) \wedge \neg(v \xrightarrow{X} w) \wedge \neg(v \xleftarrow{X} w)$ □

X.**stamp** assigns timestamps to each event of $\boldsymbol{H}$. It is normally expressed as a series of rules for updating the logical clock of a site before assigning a timestamp to an event of $\boldsymbol{H}$. When it is clear from the context, we just use $X(\mathbf{a})$ instead of X.**stamp**$(\mathbf{a}), \forall\, \mathbf{a} \in \boldsymbol{H}$. If $\mathbf{a}$ is **send**(M), it is assumed that message M carries the timestamp $X(\mathbf{a})$.

Since $\xrightarrow{X}$ is irreflexive, it is easy to see that the relations $\xrightarrow{X}$, $\xleftarrow{X}$, $\overset{X}{=}$, and $\overset{X}{\parallel}$ are mutually disjoint. Their purpose is to reflect causality, equality and concurrency from the point of view of X. Even though these relations are defined over timestamps, we overload their definition to allow them to directly compare events in $\boldsymbol{H}$. For instance, consider $\mathbf{a}, \mathbf{b} \in \boldsymbol{H}$ with timestamps $X(\mathbf{a})$ and $X(\mathbf{b})$, respectively. X reports the causal relationship (not necessarily correct) between $\mathbf{a}$ and $\mathbf{b}$, in this way:

$\mathbf{a} \overset{X}{=} \mathbf{b} \Leftrightarrow X(\mathbf{a}) \overset{X}{=} X(\mathbf{b}) \Leftrightarrow X$ "believes" that $\mathbf{a}$ and $\mathbf{b}$ are the same event.
$\mathbf{a} \xrightarrow{X} \mathbf{b} \Leftrightarrow X(\mathbf{a}) \xrightarrow{X} X(\mathbf{b}) \Leftrightarrow X$ "believes" that $\mathbf{a}$ causally precedes $\mathbf{b}$.
$\mathbf{a} \xleftarrow{X} \mathbf{b} \Leftrightarrow X(\mathbf{a}) \xleftarrow{X} X(\mathbf{b}) \Leftrightarrow X$ "believes" that $\mathbf{b}$ causally precedes $\mathbf{a}$.
$\mathbf{a} \overset{X}{\parallel} \mathbf{b} \Leftrightarrow X(\mathbf{a}) \overset{X}{\parallel} X(\mathbf{b}) \Leftrightarrow X$ "believes" that $\mathbf{a}$ and $\mathbf{b}$ are concurrent.

As an example, using the tests presented in Definition 2, we can redefine vector clocks in the form of TSS $V = (\langle \boldsymbol{S}, \xrightarrow{V} \rangle, V.\textbf{stamp})$, where:

- $\boldsymbol{S}$ is a set of N-dimensional vectors of integers.
- V.**stamp**: defined with rules **V0**, **V1** and **V2** of Sect. 2.
- Let $\mathbf{V}$ and $\mathbf{W} \in S$, then

$$\mathbf{V} \overset{V}{=} \mathbf{W} \Leftrightarrow \mathbf{V} = \mathbf{W}$$
$$\mathbf{V} \xrightarrow{V} \mathbf{W} \Leftrightarrow \mathbf{V} < \mathbf{W}$$
$$\mathbf{V} \xleftarrow{V} \mathbf{W} \Leftrightarrow \mathbf{W} \xrightarrow{V} \mathbf{V} \Leftrightarrow \mathbf{V} > \mathbf{W}$$
$$\mathbf{V} \overset{V}{\parallel} \mathbf{W} \Leftrightarrow \mathbf{V} \parallel \mathbf{W}$$

Definition 4. A TSS $X = (\langle S, \xrightarrow{X} \rangle, X.\textbf{stamp})$ *characterizes causality* [25] if $\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H}$:

$$\mathbf{a} = \mathbf{b} \Leftrightarrow \mathbf{a} \overset{X}{=} \mathbf{b}$$
$$\mathbf{a} \longrightarrow \mathbf{b} \Leftrightarrow \mathbf{a} \xrightarrow{X} \mathbf{b}$$
$$\mathbf{a} \parallel \mathbf{b} \Leftrightarrow \mathbf{a} \overset{X}{\parallel} \mathbf{b}$$

□

Notice that this is equivalent to the strong clock condition. Evidently, if TSS X characterizes causality, then $\mathbf{a} \longleftarrow \mathbf{b} \Leftrightarrow \mathbf{a} \xleftarrow{X} \mathbf{b}$.

Theorem 1. *TSS V characterizes causality.*

Proof. This result follows from property **(1)** and it was proved by Mattern [19]. □

Definition 5. A TSS $P = (\langle \boldsymbol{S}, \xrightarrow{P} \rangle, P.\textbf{stamp})$ is *plausible* if $\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H}$:

$$\mathbf{a} = \mathbf{b} \Leftrightarrow \mathbf{a} \overset{P}{=} \mathbf{b}$$
$$\mathbf{a} \longrightarrow \mathbf{b} \Rightarrow \mathbf{a} \xrightarrow{P} \mathbf{b}$$

□

A plausible TSS satisfies the weak clock condition. In addition it assigns unique timestamps to events. Just for clarity, notice that $\mathbf{a} \longleftarrow \mathbf{b} \Rightarrow \mathbf{a} \xleftarrow{P} \mathbf{b}$.

Theorem 2. *If a TSS P is plausible, then* $\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H}$, $\mathbf{a} \overset{P}{\parallel} \mathbf{b} \Rightarrow \mathbf{a} \parallel \mathbf{b}$.

Proof. If the actual causal relation were $\mathbf{a} = \mathbf{b}$, $\mathbf{a} \longrightarrow \mathbf{b}$ or $\mathbf{a} \longleftarrow \mathbf{b}$, it would have been reported as $\mathbf{a} \overset{P}{=} \mathbf{b}$, $\mathbf{a} \xrightarrow{P} \mathbf{b}$ or $\mathbf{a} \xleftarrow{P} \mathbf{b}$, respectively. Therefore, if P reports $\mathbf{a} \overset{P}{\parallel} \mathbf{b}$, then the only possibility left is $\mathbf{a} \parallel \mathbf{b}$. □

A plausible TSS P never confuses the direction of causality between any two ordered events. If in fact $\mathbf{a}$ causally precedes $\mathbf{b}$, P will always report $\mathbf{a} \xrightarrow{P} \mathbf{b}$, or if $\mathbf{b}$ causally precedes $\mathbf{a}$, P will always report $\mathbf{a} \xleftarrow{P} \mathbf{b}$. If P states that $\mathbf{a} \overset{P}{\parallel} \mathbf{b}$, this necessarily is correct. In a plausible TSS, the timestamps assigned to events are unique. Vector clocks are plausible clocks, but not every plausible TSS P characterizes causality since it is possible that $\mathbf{a} \parallel \mathbf{b}$, but instead P reports $\mathbf{a} \xrightarrow{P} \mathbf{b}$ or $\mathbf{a} \xleftarrow{P} \mathbf{b}$.

4 Combination of TSSs

Given two plausible clocks A and B, they can be easily combined to design a new plausible clock X. The objective of this combination is to produce TSSs where the ordering between more pairs of events is correctly established.

Definition 6. Let $A = (\langle \boldsymbol{S}_A, \xrightarrow{A} \rangle, A.\textbf{stamp})$ and $B = (\langle \boldsymbol{S}_B, \xrightarrow{B} \rangle, B.\textbf{stamp})$ be two plausible clocks. We define TSS $X = (\langle \boldsymbol{S}_X, \xrightarrow{X} \rangle, X.\textbf{stamp})$, where:

- $\boldsymbol{S}_X = \boldsymbol{S}_A \times \boldsymbol{S}_B$, i.e., the elements of $\boldsymbol{S}_X$ are of the form $(\mathbf{v}_A, \mathbf{v}_B)$ with $\mathbf{v}_A \in \boldsymbol{S}_A$ and $\mathbf{v}_B \in \boldsymbol{S}_B$.
- $(\mathbf{v}_A, \mathbf{v}_B) \xrightarrow{X} (\mathbf{w}_A, \mathbf{w}_B) \Leftrightarrow (\mathbf{v}_A \xrightarrow{A} \mathbf{w}_A) \wedge (\mathbf{v}_B \xrightarrow{B} \mathbf{w}_B)$
- $\forall\, \mathbf{a} \in \boldsymbol{H}$: $X.\textbf{stamp}(\mathbf{a}) = (A.\textbf{stamp}(\mathbf{a}),\ B.\textbf{stamp}(\mathbf{a}))$

We say that X is a *combination* of A and B. □

Given an arbitrary TSS $P = (\langle \boldsymbol{S}, \xrightarrow{P} \rangle, P.\textbf{stamp})$, we use the generic relations $\overset{P}{\phi}$ and $\overset{P}{\theta}$ with $\phi, \theta \in \{\longrightarrow, \longleftarrow, =, \parallel\}$ to denote one of the possible relations $\xrightarrow{P}, \xleftarrow{P}, \overset{P}{=}$ or $\overset{P}{\parallel}$. Using this notation, and considering Definition 3 and Definition 6, we have that if $\mathbf{v} = (\mathbf{v}_A, \mathbf{v}_B)$ and $\mathbf{w} = (\mathbf{w}_A, \mathbf{w}_B)$ are elements of $\boldsymbol{S}_X$ such that $\mathbf{v}_A \overset{A}{\theta} \mathbf{w}_A$ and $\mathbf{v}_B \overset{B}{\phi} \mathbf{w}_B$, then:

$$\mathbf{v} \overset{X}{=} \mathbf{w} \Leftrightarrow (\theta = \phi = \text{`} = \text{'})$$
$$\mathbf{v} \xrightarrow{X} \mathbf{w} \Leftrightarrow (\theta = \phi = \text{`} \longrightarrow \text{'})$$
$$\mathbf{v} \xleftarrow{X} \mathbf{w} \Leftrightarrow (\theta = \phi = \text{`} \longleftarrow \text{'})$$
$$\mathbf{v} \overset{X}{\parallel} \mathbf{w} \Leftrightarrow (\theta \neq \phi) \vee (\theta = \phi = \text{`} \parallel \text{'})$$

This notation brings out the nature of TSS X: if A and B disagree on the causal relationships between the same pair of events (i.e., $\theta \neq \phi$), X reports that these two events are concurrent, otherwise X just reports the same causal relationship that is reported by A and B.

Theorem 3. ***(*Rule of Contradiction in TSSs***) *Let A and B be two plausible TSSs. $\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H}$ such that $\mathbf{a} \overset{A}{\theta} \mathbf{b}$ and $\mathbf{a} \overset{B}{\phi} \mathbf{b}$, it holds that*

$$(\theta \neq \phi) \vee (\theta = \phi = \text{`} \parallel \text{'}) \Rightarrow \mathbf{a} \parallel \mathbf{b}$$

Proof. It is known that A and B are plausible TSSs, then from Definition 5:

$$(\theta \neq \phi) \wedge (\mathbf{a} = \mathbf{b}) \Rightarrow \text{Contradiction.}$$
$$(\theta \neq \phi) \wedge (\mathbf{a} \longrightarrow \mathbf{b}) \Rightarrow \text{Contradiction.}$$
$$(\theta \neq \phi) \wedge (\mathbf{a} \longleftarrow \mathbf{b}) \Rightarrow \text{Contradiction.}$$

Thus, if two plausible TSSs disagree on the causal relation between two events $\mathbf{a}$ and $\mathbf{b}$, then, necessarily, these events are concurrent. Finally, if $\theta = \phi = \text{`} \parallel \text{'}$, Theorem 2 proves that $\mathbf{a} \parallel \mathbf{b}$. □

Definition 7. Let X be an arbitrary TSS and V be a TSS based on vector clocks as defined in Sect. 3. For a finite history $\boldsymbol{H}$, we define the parameter $\rho(X)$ in this way:

$$\rho(X) = \frac{|\{\langle \mathbf{a}, \mathbf{b} \rangle \in \boldsymbol{H} \times \boldsymbol{H} : \mathbf{a} \overset{X}{\theta} \mathbf{b} \wedge \mathbf{a} \overset{V}{\phi} \mathbf{b} \wedge \theta \neq \phi\}|}{|\boldsymbol{H} \times \boldsymbol{H}|}$$

This ratio, called the *rate of errors* of X, is the proportion of all the pairs of events in the global history $\boldsymbol{H}$ whose causality relation is wrongly established by the TSS X. In particular, if X is plausible then $\rho(X)$ represents the number of pairs of concurrent events that are ordered by X divided by the total number of pairs of events. When it is clear from the context which TSS is X, we just use ρ. A more accurate TSS has a lower value of ρ (for vector clocks, ρ is 0.0). □

Theorem 4. (Plausible + Plausible = Plausible) *Let A and B be two plausible TSSs. If X is the* **combination** *of A and B then X is a plausible TSS and* $(\rho(A) \geq \rho(X)) \wedge (\rho(B) \geq \rho(X))$.

Proof. From Theorem 3 and Definition 6, it is evident that X inherits the properties of A and B concerning the detection of the cases $\mathbf{a} = \mathbf{b}$, $\mathbf{a} \longrightarrow \mathbf{b}$ and $\mathbf{a} \longleftarrow \mathbf{b}$, $\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H}$. Therefore X is plausible. Because of the same reason, the parameter $\rho(X)$ cannot be greater than either $\rho(A)$ or $\rho(B)$. It is possible (and desirable) that X detects more pairs of concurrent events than either A or B. □

In order to generalize the concept of combination of plausible clocks for more than 2 TSSs, we can notice the interesting correspondence between a *product* of ordered sets as defined in [8] and the combination of plausible clocks. Let $\langle O_1, \leq_1 \rangle, \ldots, \langle O_\mathrm{n}, \leq_\mathrm{n} \rangle$ be n ordered sets. The *product* of these ordered sets is the ordered set $\langle O_1 \times \ldots \times O_\mathrm{n}, \leq_\times \rangle$, where $\leq_\times$ is the coordinatewise order defined as:

$$(x_1, \ldots, x_\mathrm{n}) \leq_\times (y_1, \ldots, y_\mathrm{n}) \Leftrightarrow \forall\, \mathrm{i} : x_\mathrm{i} \leq_\mathrm{i} y_\mathrm{i}$$

Definition 8. Let $P_1 = (\langle \boldsymbol{S}_1, \xrightarrow{1} \rangle, P_1.\mathbf{stamp}), \ldots, P_\mathrm{n} = (\langle \boldsymbol{S}_\mathrm{n}, \xrightarrow{\mathrm{n}} \rangle, P_\mathrm{n}.\mathbf{stamp})$ be n plausible clocks. We define TSS $P_\times = (\langle S_\times, \xrightarrow{\times} \rangle, P_\times.\mathbf{stamp})$, where:

- $S_\times = S_1 \times \ldots \times S_\mathrm{n}$
- $(\mathbf{v}_1, \ldots, \mathbf{v}_\mathrm{n}) \xrightarrow{\times} (\mathbf{w}_1, \ldots, \mathbf{w}_\mathrm{n}) \Leftrightarrow \forall\, \mathrm{i} : \mathbf{v}_\mathrm{i} \xrightarrow{\mathrm{i}} \mathbf{w}_\mathrm{i}$
- $\forall\, \mathbf{a} \in H : P_\times.\mathbf{stamp}(\mathbf{a}) = (P_1.\mathbf{stamp}(\mathbf{a}), \ldots, P_\mathrm{n}.\mathbf{stamp}(\mathbf{a}))$

We say that $P_\times$ is a *combination* of $P_1, \ldots, P_\mathrm{n}$. □

5 Examples of Constant Size Clocks

We consider three groups of plausible clocks (*R-entries vector, K-Lamport clocks* and *Combined TSS*). The first one is a variant of the standard vector clocks where the vectors have a fixed number of entries. The second group is an extension of Lamport clocks, where each site keeps its logical clock and a collection of the maximum message timestamps received by itself and by sites that directly or indirectly have communicated with this site. The third group combines TSSs from the previous two groups. Obviously, these are not the only possible plausible clocks, but they are efficient and simple to implement.

5.1 R-Entries Vector TSS

R-Entries Vector TSS (REV) is a variant of vector clocks, where vectors have a fixed size R ≤ N, independent of the number of sites in the distributed system. Since there are fewer entries in the vector than sites in the system, sites will share entries in the vector. Many mappings between sites and entries of the vector are possible, however for the purposes of this paper we limit ourselves to a modulo R mapping, i.e. site i updates entry i modulo R of the vector. A similar technique is proposed by Haban and Weigel [14], where processes that are executed at the same site share an entry in the vector clock; *REV* does not have this restriction and allows processes running on different sites to share an entry in the vector. The mechanisms for timestamp comparison are almost identical to the ones used by vector clocks.

Let us define $REV = (\langle S, \xrightarrow{REV} \rangle, REV.\mathbf{stamp})$, where:

- S is a set of pairs of the form $\langle i, \mathbf{V}_i \rangle$ where i is an integer that identifies each site of the system ($0 \leq i \leq N-1$), and $\mathbf{V}_i$ is a R-dimensional vector of integers.
- $REV.\mathbf{stamp}$ is defined with the rules:
  RV0) Initial value:

  i = Unique site identification;
  $0 \leq \mathrm{j} \leq \mathrm{R} - 1 : \mathbf{V}_i[\mathrm{j}] = 0$;

  RV1) Before an event (other than a **receive**) is generated at site i:

  $\mathbf{V}_i[i \text{ modulo R}] = \mathbf{V}_i[i \text{ modulo R}] + 1$;

  RV2) When a message with timestamp $\langle s, \mathbf{V}_s \rangle$ is received at site i:

  $0 \leq \mathrm{j} \leq \mathrm{R} - 1 : \mathbf{V}_i[\mathrm{j}] = \max(\mathbf{V}_i[\mathrm{j}], \mathbf{V}_s[\mathrm{j}])$
  $\mathbf{V}_i[i \text{ modulo R}] = \mathbf{V}_i[i \text{ modulo R}] + 1$;
- Let $\langle i, \mathbf{V}_i \rangle, \langle j, \mathbf{V}_j \rangle \in S$, then:

$$\begin{aligned}\langle i, \mathbf{V}_i \rangle \xrightarrow{REV} \langle j, \mathbf{V}_j \rangle \\ \Leftrightarrow (i = j \wedge \mathbf{V}_i[i \text{ modulo R}] \\ < \mathbf{V}_j[j \text{ modulo R}]) \vee \\ (i \neq j \wedge \mathbf{V}_i < \mathbf{V}_j \wedge \mathbf{V}_i[j \text{ modulo R}] \\ < \mathbf{V}_j[j \text{ modulo R}])\end{aligned}$$

Vectors $\mathbf{V}_i$ and $\mathbf{V}_j$ are compared using the tests presented in Definition 2, with the only difference that they have R entries instead of N. For completeness, notice that:

$$\begin{aligned}\langle i, \mathbf{V}_i \rangle \overset{REV}{=} \langle j, \mathbf{V}_j \rangle \Leftrightarrow (i = j \wedge \mathbf{V}_i[i \text{ modulo R}] \\ = \mathbf{V}_j[j \text{ modulo R}])\end{aligned}$$

$$\begin{aligned}\langle i, \mathbf{V}_i \rangle \xleftarrow{REV} \langle j, \mathbf{V}_j \rangle \Leftrightarrow (i = j \wedge \mathbf{V}_i[i \text{ modulo R}] \\ > \mathbf{V}_j[j \text{ modulo R}]) \vee \\ (i \neq j \wedge \mathbf{V}_i > \mathbf{V}_j \wedge \mathbf{V}_i[i \text{ modulo R}] \\ > \mathbf{V}_j[i \text{ modulo R}])\end{aligned}$$

$$\begin{aligned}\langle i, \mathbf{V}_i \rangle \overset{REV}{\parallel} \langle j, \mathbf{V}_j \rangle \Leftrightarrow i \neq j \wedge \neg(\langle i, \mathbf{V}_i \rangle \xrightarrow{REV} \langle j, \mathbf{V}_j \rangle) \\ \wedge \neg(\langle i, \mathbf{V}_i \rangle \xleftarrow{REV} \langle j, \mathbf{V}_j \rangle)\end{aligned}$$

Theorem 5. *REV is a plausible TSS.*

Proof. We have to prove, first, that REV is a TSS and, second, that REV is plausible. The former is proved by

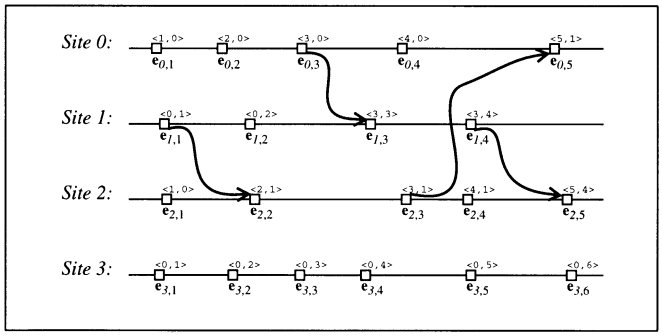

Fig. 2. Execution timestamped with REV (R = 2)

verifying that the relation $\stackrel{REV}{\longrightarrow}$ is irreflexive and transitive, and the latter is proved by confirming that REV satisfies Definition 5.

Let $\langle i, \mathbf{V}_i\rangle$, $\langle j, \mathbf{V}_j\rangle$, $\langle k, \mathbf{V}_k\rangle \in S$. Evidently, $\neg(\langle i, \mathbf{V}_i\rangle \stackrel{REV}{\longrightarrow} \langle i, \mathbf{V}_i\rangle)$, thus, $\stackrel{REV}{\longrightarrow}$ is irreflexive. Let us assume that $\langle i, \mathbf{V}_i\rangle \stackrel{REV}{\longrightarrow} \langle j, \mathbf{V}_j\rangle$ and $\langle j, \mathbf{V}_j\rangle \stackrel{REV}{\longrightarrow} \langle k, \mathbf{V}_k\rangle$. If $i = j = k$, it is clear that $\langle i, \mathbf{V}_i\rangle \stackrel{REV}{\longrightarrow} \langle k, \mathbf{V}_k\rangle$. Now, if at least one of the timestamps corresponds to a different site, the comparison of vectors as presented in Definition 2 guarantees that $\langle i, \mathbf{V}_i\rangle \stackrel{REV}{\longrightarrow} \langle k, \mathbf{V}_k\rangle$. Hence, $\stackrel{REV}{\longrightarrow}$ is transitive and, therefore, REV is a TSS.

Since the timestamps of REV include a site identification, no two events occurring at different sites receive the same timestamp. Similarly, rules **RV1** and **RV2** guarantee that no two different events occurring at the same site receive the same timestamp. Therefore:

$$\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H} : \mathbf{a} = \mathbf{b} \Leftrightarrow \mathbf{a} \stackrel{REV}{=} \mathbf{b}$$

Let $\mathbf{a}, \mathbf{b} \in \boldsymbol{H}$ be two distinct events such that $REV(\mathbf{a}) = \langle i, \mathbf{V}_i\rangle$ and $REV(\mathbf{b}) = \langle j, \mathbf{V}_j\rangle$. If **a** and **b** were executed at the same site, just by comparing $\mathbf{V}_i[i$ modulo R] and $\mathbf{V}_j[j$ modulo R], we establish the order of these events. Now, consider the case where **a** and **b** were generated at different sites. Because of the definition of REV.**stamp**, if $\mathbf{a} \longrightarrow \mathbf{b}$ then necessarily $\mathbf{V}_i < \mathbf{V}_j$. Besides, $\mathbf{V}_j[j$ modulo R] must be strictly greater than $\mathbf{V}_i[j$ modulo R] because $\mathbf{V}_j[j$ modulo R] is incremented when **b** is executed. Thus, if $\mathbf{V}_i < \mathbf{V}_j$ but $\mathbf{V}_i[j$ modulo R] = $\mathbf{V}_j[j$ modulo R], then **a** and **b** are concurrent events. Finally, if $i \neq j$ and $\mathbf{V}_i = \mathbf{V}_j$ or $\mathbf{V}_i \parallel \mathbf{V}_j$ then necessarily $\mathbf{a} \parallel \mathbf{b}$. Therefore:

$$\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H} : \mathbf{a} \longrightarrow \mathbf{b} \Rightarrow \mathbf{a} \stackrel{REV}{\longrightarrow} \mathbf{b}$$

In conclusion, REV is a plausible TSS. □

Figure 2 shows the same execution presented in Fig. 1, but using timestamps from REV (R = 2). In order to simplify, the part of the timestamp corresponding to the site identification has been omitted. This clock establishes correctly the causal relationship between 320 out of the 400 possible pairs of events, i.e. $\rho(REV)$ is 0.2 for this particular history. For instance, REV recognizes that $\mathbf{e}_{3,1} \parallel \mathbf{e}_{0,2}$, but orders concurrent events when it reports $\mathbf{e}_{3,1} \stackrel{REV}{\longrightarrow} \mathbf{e}_{1,2}$, since $\mathbf{e}_{3,1} \parallel \mathbf{e}_{1,2}$.

5.2 K-Lamport Time Stamping System

This family of TSSs uses the same data structures as REV: a site identification and a vector of integers. However, the rules for updating this vector are different. Each site keeps a Lamport clock together with the maximum timestamp of any message received by itself and by the K-2 previous sites that directly or indirectly have had communications with this site.

As an intuition for this clock, consider a case where two events **a** and **b** were executed at different sites and have Lamport clock timestamps 7 and 10, respectively. With just these timestamps we would conclude that $\mathbf{a} \longrightarrow \mathbf{b}$. Now, let's assume that each site keeps the maximum timestamp of all the received messages in a local variable h. If h is 5 when **b** is executed, we know that **a** can not causally precede **b**, because otherwise h would have been updated and it would be greater than or equal to 7. Therefore **a** and **b** are concurrent events. Notice that this mechanism can be extended if site i keeps not only its Lamport clock and the maximum timestamp received, but also the maximum timestamp received by any site that has sent a message to site i, and so on. This extra information will give us the ability to discern more accurately when it is true that two events are actually ordered.

In order to better understand the dynamics of the *K-Lamport TSS*, we present and analyze the properties of the basic case *2-Lamport TSS*.

2-Lamport Time Stamping System

2-Lamport TSS (2LA) is an extension of Lamport clocks where site i has a 2-entries vector $\mathbf{V}_i$. Site i keeps a local Lamport Clock in $\mathbf{V}_i[0]$ and saves the maximum timestamp carried by any received message in $\mathbf{V}_i[1]$.

We define $2LA = (\langle S, \stackrel{2LA}{\longrightarrow}\rangle, 2LA.\textbf{stamp})$, where:

- S is a set of pairs of the form $\langle i, \mathbf{V}_i\rangle$ where i is an integer that identifies each site of the system ($0 \leq i \leq N-1$), and $\mathbf{V}_i$ is a 2-dimensional vector of integers.
- $2LA$.**stamp** is defined with the rules:

  2L0) Initial value:

  i = Unique site identification;
  $\mathbf{V}_i[0] = 0$;
  $\mathbf{V}_i[1] = 0$;

  2L1) Before an event (other than a **receive**) is generated:

  $\mathbf{V}_i[0] = \mathbf{V}_i[0] + 1$;

  2L2) When a message with timestamp $\langle s, \mathbf{V}_s\rangle$ is received:

  $\mathbf{V}_i[0] = \max(\mathbf{V}_i[0], \mathbf{V}_s[0])$;
  $\mathbf{V}_i[0] = \mathbf{V}_i[0] + 1$;
  $\mathbf{V}_i[1] = \max(\mathbf{V}_i[1], \mathbf{V}_s[0])$;

- Let $\langle i, \mathbf{V}_i\rangle$, $\langle j, \mathbf{V}_j\rangle \in S$, then:

$$\langle i, \mathbf{V}_i\rangle \stackrel{2LA}{\longrightarrow} \langle j, \mathbf{V}_j\rangle \Leftrightarrow (i = j \wedge \mathbf{V}_i[0] < \mathbf{V}_j[0]) \vee (i \neq j \wedge \mathbf{V}_i[0] \leq \mathbf{V}_j[1])$$

Notice that $\mathbf{V}_i[0] > \mathbf{V}_i[1]$ for any timestamp assigned to any event. For completeness, we have that:

$$\langle i, \mathbf{V}_i\rangle \overset{2LA}{=} \langle j, \mathbf{V}_j\rangle \Leftrightarrow (i = j \wedge \mathbf{V}_i[0] = \mathbf{V}_j[0])$$

$$\langle i, \mathbf{V}_i\rangle \overset{2LA}{\longleftarrow} \langle j, \mathbf{V}_j\rangle \Leftrightarrow (i = j \wedge \mathbf{V}_i[0] > \mathbf{V}_j[0]) \vee (i \neq j \wedge \mathbf{V}_i[1]\mathbf{V}_j[0])$$

$$\langle i, \mathbf{V}_i\rangle \overset{2LA}{\parallel} \langle j, \mathbf{V}_j\rangle \Leftrightarrow i \neq j \wedge \neg(\langle i, \mathbf{V}_i\rangle \overset{2LA}{\longrightarrow} \langle j, \mathbf{V}_j\rangle) \wedge \neg(\langle i, \mathbf{V}_i\rangle \overset{2LA}{\longleftarrow} \langle j, \mathbf{V}_j\rangle)$$

Theorem 6. *$2LA$ is a plausible TSS.*

Proof. We have to prove that $\overset{2LA}{\longrightarrow}$ is irreflexive and transitive, and that $2LA$ satisfies Definition 5.

Let $\langle i, \mathbf{V}_i\rangle$, $\langle j, \mathbf{V}_j\rangle$, $\langle k, \mathbf{V}_k\rangle \in S$. Since $\neg(\langle i, \mathbf{V}_i\rangle \overset{2LA}{\longrightarrow} \langle i, \mathbf{V}_i\rangle)$, $\overset{2LA}{\longrightarrow}$ is irreflexive. Let us assume that $\langle i, \mathbf{V}_i\rangle \overset{2LA}{\longrightarrow} \langle j, \mathbf{V}_j\rangle$ and $\langle j, \mathbf{V}_j\rangle \overset{2LA}{\longrightarrow} \langle k, \mathbf{V}_k\rangle$. We can see that:

$$(i = j = k) \Rightarrow (\mathbf{V}_i[0] < \mathbf{V}_j[0] < \mathbf{V}_k[0])$$
$$\Rightarrow (i = k \wedge \mathbf{V}_i[0] < \mathbf{V}_k[0]) \Rightarrow \langle i, \mathbf{V}_i\rangle \overset{2LA}{\longrightarrow} \langle k, \mathbf{V}_k\rangle$$
$$(i \neq j = k) \Rightarrow (\mathbf{V}_i[0] \leq \mathbf{V}_j[1] \leq \mathbf{V}_k[1])$$
$$\Rightarrow (i \neq k \wedge \mathbf{V}_i[0] \leq \mathbf{V}_k[1]) \Rightarrow \langle i, \mathbf{V}_i\rangle \overset{2LA}{\longrightarrow} \langle k, \mathbf{V}_k\rangle$$
$$(j \neq i = k) \Rightarrow (\mathbf{V}_i[0] \leq \mathbf{V}_j[1] < \mathbf{V}_j[0] \leq \mathbf{V}_k[1] < \mathbf{V}_k[0])$$
$$\Rightarrow (i = k \wedge \mathbf{V}_i[0] < \mathbf{V}_k[0]) \Rightarrow \langle i, \mathbf{V}_i\rangle \overset{2LA}{\longrightarrow} \langle k, \mathbf{V}_k\rangle$$
$$(k \neq i = j) \Rightarrow (\mathbf{V}_i[0] < \mathbf{V}_j[0] \leq \mathbf{V}_k[1])$$
$$\Rightarrow (i \neq k \wedge \mathbf{V}_i[0] \leq \mathbf{V}_k[1]) \Rightarrow \langle i, \mathbf{V}_i\rangle \overset{2LA}{\longrightarrow} \langle k, \mathbf{V}_k\rangle$$
$$(i \neq j \neq k) \Rightarrow (\mathbf{V}_i[0] \leq \mathbf{V}_j[1] < \mathbf{V}_j[0] \leq \mathbf{V}_k[1])$$
$$\Rightarrow (i \neq k \wedge \mathbf{V}_i[0] \leq \mathbf{V}_k[1]) \Rightarrow \langle i, \mathbf{V}_i\rangle \overset{2LA}{\longrightarrow} \langle k, \mathbf{V}_k\rangle$$

Thus, $\overset{2LA}{\longrightarrow}$ is transitive and therefore $2LA$ is a TSS.

From rules **2L1** and **2L2**, and the fact that the timestamps generated by $2LA$ include a site identification, it is easy to see that:

$$\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H} : \mathbf{a} = \mathbf{b} \Leftrightarrow \mathbf{a} \overset{2LA}{=} \mathbf{b}$$

Let $\mathbf{a}, \mathbf{b} \in \boldsymbol{H}$ be two arbitrary events such that $2LA(\mathbf{a}) = \langle i, \mathbf{V}_i\rangle$ and $2LA(\mathbf{b}) = \langle j, \mathbf{V}_j\rangle$. If **a** and **b** occur at the same site the causal relation is correctly established just by comparing $\mathbf{V}_i[0]$ and $\mathbf{V}_j[0]$. Consider the case where **a** and **b** have been executed at different sites. If **a** is **send**(M) and **b** is **receive**(M), then $\mathbf{V}_i[0]$ would have been communicated to the site where **b** occurs and the entry $\mathbf{V}_j[1]$ would have been updated. Therefore:

$$\mathbf{a} = \textbf{send}(\mathrm{M}) \wedge \mathbf{b} = \textbf{receive}(\mathrm{M}) \Rightarrow (\mathbf{V}_i[0] \leq \mathbf{V}_j[1])$$

These conclusions can be easily generalized when **a** and **b** are causally related:

$$\mathbf{a} \longrightarrow \mathbf{b} \Rightarrow (\mathbf{V}_i[0] \leq \mathbf{V}_j[1])$$

On the other hand, if $\mathbf{V}_i[0] > \mathbf{V}_j[1]$ then **a** does not causally precede **b** because of the definition of $2LA$.**stamp**. Conversely, if $\mathbf{V}_j[0] > \mathbf{V}_i[1]$, **b** does not causally precede **a**. Therefore,

$$(\mathbf{V}_i[0] > \mathbf{V}_j[1]) \wedge (\mathbf{V}_j[0] > \mathbf{V}_i[1]) \Rightarrow \mathbf{a} \parallel \mathbf{b}$$

Table 1. Possible relations between $\mathbf{V}_i$ and $\mathbf{V}_j$ ($2LA$).

	$\mathbf{V}_i[0] \leq \mathbf{V}_j[1]$	$\mathbf{V}_i[0] > \mathbf{V}_j[1]$
$\mathbf{V}_j[0] \leq \mathbf{V}_i[1]$	Impossible	$\mathbf{a} \longleftarrow \mathbf{b}$ *uncertain*
$\mathbf{V}_j[0] > \mathbf{V}_i[1]$	$\mathbf{a} \longrightarrow \mathbf{b}$ *uncertain*	$\mathbf{a} \parallel \mathbf{b}$ *certain!*

Since we know that $(\mathbf{V}_i[1] < \mathbf{V}_i[0]) \wedge (\mathbf{V}_j[1] < \mathbf{V}_j[0])$, we can notice that:

$$(\mathbf{V}_i[0] \leq \mathbf{V}_j[1]) \wedge (\mathbf{V}_j[0] \leq \mathbf{V}_i[1]) \Rightarrow$$
$$\mathbf{V}_i[0] \leq \mathbf{V}_j[1] < \mathbf{V}_j[0] \leq \mathbf{V}_i[1] \Rightarrow$$
$$\mathbf{V}_i[0] < \mathbf{V}_i[1] \Rightarrow \text{Contradiction.}$$

Table 1 summarizes the previous relations. Using the information in this table, $2LA$ detects correctly all the cases where $\mathbf{a} \longrightarrow \mathbf{b}$ or $\mathbf{a} \longleftarrow \mathbf{b}$. Therefore,

$$\mathbf{a} \longrightarrow \mathbf{b} \Rightarrow \mathbf{a} \overset{2LA}{\longrightarrow} \mathbf{b}$$

Thus, $2LA$ is a plausible TSS. □

General Case: K-Lamport TSS

Consider briefly what *3-Lamport TSS (3LA)* would be. Timestamps are of the form $\langle i, \mathbf{V}_i\rangle$ where i is a site identification and $\mathbf{V}_i$ is a 3 entry vector, whose entry 0 is a Lamport clock. Every message sent by site i carries the timestamp $\langle i, \mathbf{V}_i\rangle$ that corresponds to the time assigned to the particular send event. When site i receives a message with timestamp $\langle s, \mathbf{V}_s\rangle$, its entry $\mathbf{V}_i[0]$ is updated with $\mathbf{V}_s[0]$ in a standard Lamport clock fashion and its entries $\mathbf{V}_i[1]$ and $\mathbf{V}_i[2]$ are max-ed with $\mathbf{V}_s[0]$ and $\mathbf{V}_s[1]$. The results shown in Table 1 for entries 0 and 1 of two arbitrary timestamps $\mathbf{V}_i$ and $\mathbf{V}_j$ are still valid for $3LA$; besides, entries 1 and 2 of $\mathbf{V}_i$ and $\mathbf{V}_j$ will exhibit these same relations. Thus, given two different events **a** and **b** with timestamps $\langle i, \mathbf{V}_i\rangle$ and $\langle j, \mathbf{V}_j\rangle$:

$$\mathbf{a} \longrightarrow \mathbf{b} \Rightarrow (\mathbf{V}_i[0] \leq \mathbf{V}_j[1]) \wedge (\mathbf{V}_i[1] \leq \mathbf{V}_j[2])$$
$$\mathbf{a} \longleftarrow \mathbf{b} \Rightarrow (\mathbf{V}_i[1] \geq \mathbf{V}_j[0]) \wedge (\mathbf{V}_i[2] \geq \mathbf{V}_j[1])$$

The relation $\overset{3LA}{\longrightarrow}$ is equivalent to $(\mathbf{V}_i[0] \leq \mathbf{V}_j[1]) \wedge (\mathbf{V}_i[1] \leq \mathbf{V}_j[2])$. With this test, $3LA$ is detecting correctly all the cases where $\mathbf{a} \longrightarrow \mathbf{b}$. If this test fails and $3LA$ encounters that $(\mathbf{V}_i[1] \geq \mathbf{V}_j[0]) \wedge (\mathbf{V}_i[2] \geq \mathbf{V}_j[1])$ holds, then $\overset{3LA}{\longleftarrow}$ is reported, detecting correctly all the cases where $\mathbf{a} \longleftarrow \mathbf{b}$. If both tests fail, $3LA$ reports $\mathbf{a} \overset{3LA}{\parallel} \mathbf{b}$.

K-Lamport TSS (KLA) is a generalization of $2LA$ and $3LA$, where we extend the pattern shown in these TSSs to a vector with K entries. Let us define $KLA = (\langle S, \overset{KLA}{\longrightarrow}\rangle, KLA.\textbf{stamp})$, where:

- S is a set of pairs of the form $\langle i, \mathbf{V}_i\rangle$ where i is an integer that identifies each site of the system ($0 \leq i \leq N-1$), and $\mathbf{V}_i$ is a K-dimensional vector of integers.
- KLA.**stamp** is defined with the rules:
  KL0) Initial value:

  $i =$ Unique site identification;

  $0 \leq \mathrm{j} \leq \mathrm{K} - 1 : \mathbf{V}_i[\mathrm{j}] = 0;$

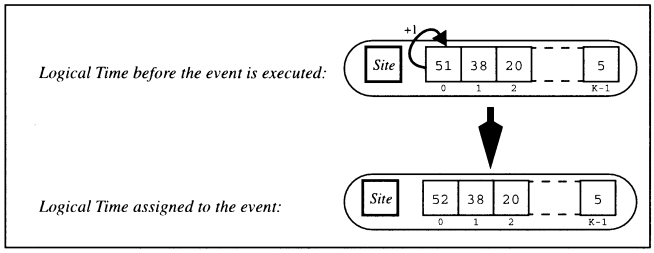

Fig. 3. KLA update rule for a local event

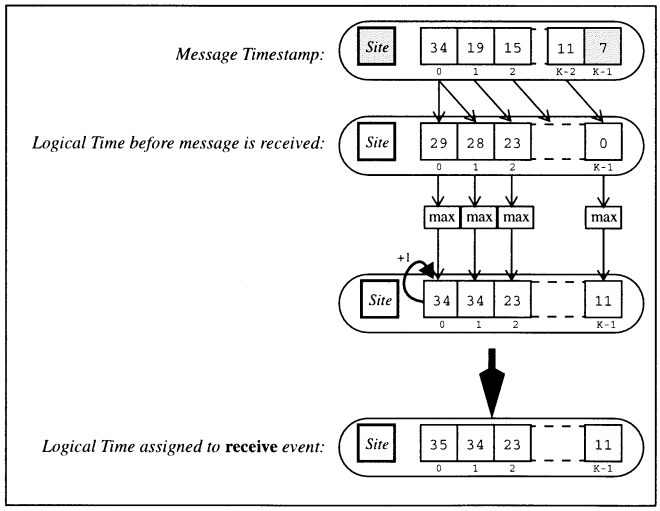

Fig. 4. KLA update rule when a message is received

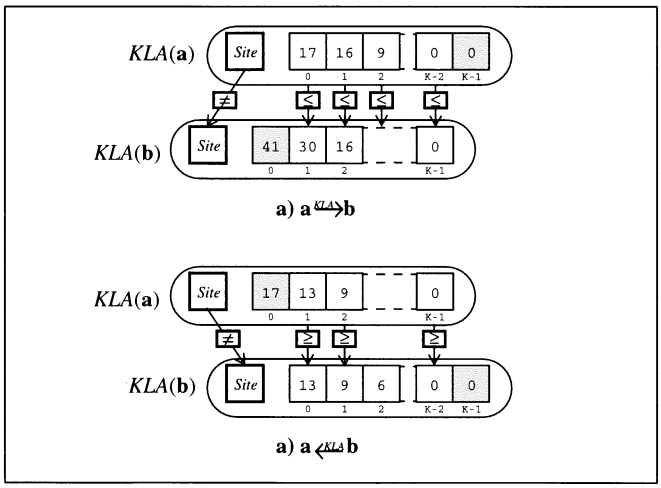

Fig. 5. Tests for causally related events under KLA

KL1) Before an event (other than a **receive**) is generated:

$\mathbf{V}_i[0] = \mathbf{V}_i[0] + 1$; (See Figure 3)

KL2) When a message with timestamp $\langle s, \mathbf{V}_s \rangle$ is received:

$\mathbf{V}_i[0] = \max(\mathbf{V}_i[0], \mathbf{V}_s[0])$;

$\mathbf{V}_i[0] = \mathbf{V}_i[0] + 1$;

$1 \leq \mathrm{j} \leq \mathrm{K} - 1 : \mathbf{V}_i[j] = \max(\mathbf{V}_i[\mathrm{j}], \mathbf{V}_s[\mathrm{j} - 1])$;

(See Figure 4)

– Let $\langle i, \mathbf{V}_i \rangle$, $\langle j, \mathbf{V}_j \rangle \in S$, then:

$$\langle i, \mathbf{V}_i \rangle \overset{KLA}{\longrightarrow} \langle j, \mathbf{V}_j \rangle \Leftrightarrow (i = j \wedge \mathbf{V}_i[0] < \mathbf{V}_j[0]) \vee (i \neq j \wedge \mathbf{V}_i[0] \leq \mathbf{V}_j[1] \wedge \mathbf{V}_i[1] \leq \mathbf{V}_j[2] \wedge \ldots \ldots \wedge \mathbf{V}_i[\mathrm{K}-2] \leq \mathbf{V}_j[\mathrm{K}-1])$$

The non-zero entries of $\mathbf{V}_i$ satisfy that for any $k < \mathrm{K} - 1$, $\mathbf{V}_i[k] > \mathbf{V}_i[k+1]$. The rightmost entries of the vector could contain zeroes (see Sect. 5.4). However, this does not affect the correctness of the algorithm. As before, notice that:

$$\langle i, \mathbf{V}_i \rangle \overset{KLA}{=} \langle j, \mathbf{V}_j \rangle \Leftrightarrow (i = j \wedge \mathbf{V}_i[0] = \mathbf{V}_j[0])$$

$$\langle i, \mathbf{V}_i \rangle \overset{KLA}{\longleftarrow} \langle j, \mathbf{V}_j \rangle \Leftrightarrow (i = j \wedge \mathbf{V}_i[0] > \mathbf{V}_j[0]) \vee (i \neq j \wedge \mathbf{V}_i[1] \geq \mathbf{V}_j[0] \wedge \mathbf{V}_i[2] \geq \mathbf{V}_j[1] \wedge \ldots \ldots \wedge \mathbf{V}_i[\mathrm{K}-1] \geq \mathbf{V}_j[\mathrm{K}-2])$$

$$\langle i, \mathbf{V}_i \rangle \overset{KLA}{\parallel} \langle j, \mathbf{V}_j \rangle \Leftrightarrow i \neq j \wedge \neg(\langle i, \mathbf{V}_i \rangle \overset{KLA}{\longrightarrow} \langle j, \mathbf{V}_j \rangle) \wedge \neg(\langle i, \mathbf{V}_i \rangle \overset{KLA}{\longleftarrow} \langle j, \mathbf{V}_j \rangle)$$

Figures 5a and 5b present the tests made by KLA when two events occur at different sites and are causally related from the point of view of KLA.

Theorem 7. *KLA is a plausible TSS.*

Proof. We have to prove that KLA is a TSS (i.e., $\overset{KLA}{\longrightarrow}$ is irreflexive and transitive), and that KLA is plausible, i.e., it satisfies Definition 5.

Let $\langle i, \mathbf{V}_i \rangle$, $\langle j, \mathbf{V}_j \rangle$, $\langle k, \mathbf{V}_k \rangle \in S$. Since $\neg(\langle i, \mathbf{V}_i \rangle \overset{KLA}{\longrightarrow} \langle i, \mathbf{V}_i \rangle)$, $\overset{KLA}{\longrightarrow}$ is irreflexive. Let us assume that $\langle i, \mathbf{V}_i \rangle \overset{KLA}{\longrightarrow} \langle j, \mathbf{V}_j \rangle$ and $\langle j, \mathbf{V}_j \rangle \overset{KLA}{\longrightarrow} \langle k, \mathbf{V}_k \rangle$. Notice that:

$$(i = j = k) \Rightarrow (\mathbf{V}_i[0] < \mathbf{V}_j[0] < \mathbf{V}_k[0]) \Rightarrow \langle i, \mathbf{V}_i \rangle \overset{KLA}{\longrightarrow} \langle k, \mathbf{V}_k \rangle$$

$$(i \neq j = k) \Rightarrow (\mathbf{V}_i[0] \leq \mathbf{V}_j[1] \leq \mathbf{V}_k[1]) \wedge (\mathbf{V}_i[1] \leq \mathbf{V}_j[2] \leq \mathbf{V}_k[2]) \wedge \ldots \ldots \wedge (\mathbf{V}_i[\mathrm{K}-2] \leq \mathbf{V}_j[\mathrm{K}-1] \leq \mathbf{V}_k[\mathrm{K}-1]) \Rightarrow \langle i, \mathbf{V}_i \rangle \overset{KLA}{\longrightarrow} \langle k, \mathbf{V}_k \rangle$$

$$(j \neq i = k) \Rightarrow (\mathbf{V}_i[0] \leq \mathbf{V}_j[1] \leq \mathbf{V}_k[2] < \mathbf{V}_k[1] < \mathbf{V}_k[0]) \Rightarrow \langle i, \mathbf{V}_i \rangle \overset{KLA}{\longrightarrow} \langle k, \mathbf{V}_k \rangle$$

$$(k \neq i = j) \Rightarrow (\mathbf{V}_i[0] < \mathbf{V}_j[0] \leq \mathbf{V}_k[1]) \wedge (\mathbf{V}_i[1] \leq \mathbf{V}_j[1] \leq \mathbf{V}_k[2]) \wedge \ldots \ldots \wedge (\mathbf{V}_i[\mathrm{K}-2] \leq \mathbf{V}_j[\mathrm{K}-2] \leq \mathbf{V}_k[\mathrm{K}-1]) \Rightarrow \langle i, \mathbf{V}_i \rangle \overset{KLA}{\longrightarrow} \langle k, \mathbf{V}_k \rangle$$

$$(i \neq j \neq k) \Rightarrow (\mathbf{V}_i[0] \leq \mathbf{V}_j[1] \leq \mathbf{V}_k[2] < \mathbf{V}_k[1]) \wedge (\mathbf{V}_i[1] \leq \mathbf{V}_j[2] \leq \mathbf{V}_k[3] < \mathbf{V}_k[2]) \wedge \ldots \ldots \wedge (\mathbf{V}_i[\mathrm{K}-2] \leq \mathbf{V}_j[\mathrm{K}-1] < \mathbf{V}_j[\mathrm{K}-2] \leq \mathbf{V}_k[\mathrm{K}-1]) \Rightarrow \langle i, \mathbf{V}_i \rangle \overset{KLA}{\longrightarrow} \langle k, \mathbf{V}_k \rangle$$

Thus, $\overset{KLA}{\longrightarrow}$ is transitive and therefore KLA is a TSS.

From rules **KL1** and **KL2**, and the fact that the timestamps generated by KLA include a site identification, it is easy to see that:

Table 2. Possible relations between $\mathbf{V}_i$ and $\mathbf{V}_j$ (KLA)

	$\mathbf{V}_i[k] \leq \mathbf{V}_j[k+1]$	$\mathbf{V}_i[k] > \mathbf{V}_j[k+1]$
$\mathbf{V}_j[k] \leq \mathbf{V}_i[k+1]$	Impossible	**a** $\longleftarrow$ **b** *uncertain*
$\mathbf{V}_j[k] > \mathbf{V}_i[k+1]$	**a** $\longrightarrow$ **b** *uncertain*	**a** $\parallel$ **b** *certain!*

$$\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H} : \mathbf{a} = \mathbf{b} \Leftrightarrow \mathbf{a} \overset{KLA}{=} \mathbf{b}$$

Let $\mathbf{a}, \mathbf{b} \in \boldsymbol{H}$ be two arbitrary events such that KLA(**a**) = $\langle i, \mathbf{V}_i \rangle$ and KLA(**b**) = $\langle j, \mathbf{V}_j \rangle$. If **a** is **send**(M) and **b** is **receive**(M), then, $\forall\, k < \mathrm{K}-1, \mathbf{V}_i[k]$ would have been communicated to the site where **b** occurs and the entry $\mathbf{V}_j[k+1]$ would have been updated. This can be easily generalized for any case where **a** and **b** are causally related. Therefore, $\forall\, k < \mathrm{K}-1$:

$$\mathbf{a} \longrightarrow \mathbf{b} \Rightarrow (\mathbf{V}_i[\mathrm{k}] \leq \mathbf{V}_j[\mathrm{k}+1])$$

If ($\mathbf{V}_i$[k] > $\mathbf{V}_j$[k+1]), **a** does not causally precede **b** because of the definition of KLA.**stamp**. Conversely, if ($\mathbf{V}_j$[k] > $\mathbf{V}_i$[k + 1]), **b** does not causally precede **a**. Therefore,

$$\exists\, \mathrm{k} < \mathrm{K}-1 \text{ such that } (\mathbf{V}_i[\mathrm{k}] > \mathbf{V}_j[\mathrm{k}+1]) \wedge (\mathbf{V}_j[\mathrm{k}] > \mathbf{V}_i[\mathrm{k}+1]) \Rightarrow \mathbf{a} \parallel \mathbf{b}$$

We know that for the non-zero entries of the vectors[2], ($\mathbf{V}_i$[k+ 1] < $\mathbf{V}_i$[k]) $\wedge$ ($\mathbf{V}_j$[k + 1] < $\mathbf{V}_j$[k]), then:

$$\begin{array}{ll} (\mathbf{V}_i[\mathrm{k}] \leq \mathbf{V}_j[\mathrm{k}+1]) \wedge (\mathbf{V}_j[\mathrm{k}] \leq \mathbf{V}_i[\mathrm{k}+1]) \Rightarrow & \\ \mathbf{V}_i[\mathrm{k}] \leq \mathbf{V}_j[\mathrm{k}+1] < \mathbf{V}_j[\mathrm{k}] \leq \mathbf{V}_i[\mathrm{k}+1] & \Rightarrow \\ \mathbf{V}_i[\mathrm{k}] < \mathbf{V}_i[\mathrm{k}+1] & \Rightarrow \text{Contradiction.} \end{array}$$

Table 2 summarizes the previous relations. Using the information on this table, KLA detects correctly all the cases where **a** $\longrightarrow$ **b** or **a** $\longleftarrow$ **b**.

Therefore,

$$\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H} : \mathbf{a} \longrightarrow \mathbf{b} \Rightarrow \mathbf{a} \overset{KLA}{\longrightarrow} \mathbf{b}$$

KLA satisfies Definition 5 and therefore it is a plausible TSS. □

Every case where **a** $\parallel$ **b** that is recognized by $(K-1)LA$, is also recognized by KLA, but the converse is not always true. Thus, KLA can provide higher ordering accuracy than $(K-1)LA$. Figure 6 shows the same execution presented in Figs. 1 and 2, but using timestamps from KLA (K = 3). This clock fails to establish the causal relationship between 46 out of 400 possible pairs of events (ρ = 0.115). For instance, it detects correctly that $\mathbf{e}_{3,1} \parallel \mathbf{e}_{1,2}$, but fails when it reports $\mathbf{e}_{1,3} \overset{KLA}{\longleftarrow} \mathbf{e}_{3,3}$, since actually $\mathbf{e}_{1,3} \parallel \mathbf{e}_{3,3}$.

5.3 Combined Time Stamping System (Comb)

By using Definition 6, we can create a combination of REV and KLA. Let us define $Comb$ = ($\langle S, \overset{Comb}{\longrightarrow} \rangle, Comb$.**stamp**), where:

[2] There is at least one non-zero entry in each of the vectors.

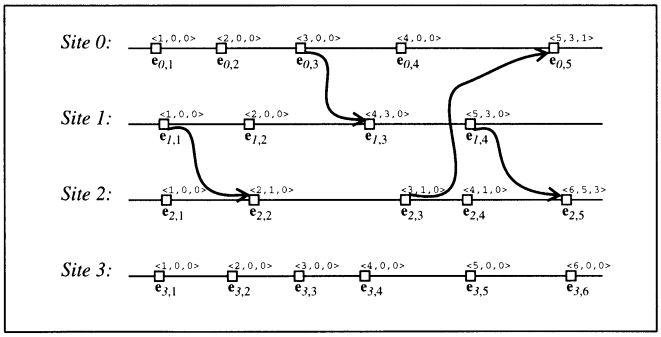

Fig. 6. Execution timestamped with KLA (K = 3)

- S is a set of elements of the form ($\langle i, \mathbf{V}_i \rangle, \langle i, \mathbf{W}_i \rangle$), where i is an integer that identifies each site of the system ($0 \leq i \leq \mathrm{N}-1$), $\mathbf{V}_i$ a R-Dimensional vector of integers and $\mathbf{W}_i$ a K-Dimensional vector of integers.[3]
- $\forall\, \mathbf{a} \in \boldsymbol{H}$: $Comb$.**stamp**(**a**) = (REV.**stamp**(**a**), KLA.**stamp**(**a**))
- Let ($\langle i, \mathbf{V}_i \rangle, \langle i, \mathbf{W}_i \rangle$), ($\langle j, \mathbf{V}_j \rangle, \langle j, \mathbf{W}_j \rangle$) $\in S$. If $\langle i, \mathbf{V}_i \rangle \overset{REV}{\theta} \langle j, \mathbf{V}_j \rangle$ and $\langle i, \mathbf{W}_i \rangle \overset{KLA}{\phi} \langle j, \mathbf{W}_j \rangle$, then:

$$(\langle i, \mathbf{V}_i \rangle, \langle i, \mathbf{W}_i \rangle) \overset{Comb}{\longrightarrow} (\langle j, \mathbf{V}_j \rangle, \langle j, \mathbf{W}_j \rangle) \Leftrightarrow (\phi = \theta = \text{`}\longrightarrow\text{'})$$

It is easy to see that:

$$(\langle i, \mathbf{V}_i \rangle, \langle i, \mathbf{W}_i \rangle) \overset{Comb}{=} (\langle j, \mathbf{V}_j \rangle, \langle j, \mathbf{W}_j \rangle) \Leftrightarrow (\phi = \theta = \text{`}=\text{'})$$

$$(\langle i, \mathbf{V}_i \rangle, \langle i, \mathbf{W}_i \rangle) \overset{Comb}{\longleftarrow} (\langle j, \mathbf{V}_j \rangle, \langle j, \mathbf{W}_j \rangle) \Leftrightarrow (\phi = \theta = \text{`}\longleftarrow\text{'})$$

$$(\langle i, \mathbf{V}_i \rangle, \langle i, \mathbf{W}_i \rangle) \overset{Comb}{\parallel} (\langle j, \mathbf{V}_j \rangle, \langle j, \mathbf{W}_j \rangle) \Leftrightarrow (\phi \neq \theta) \vee (\theta = \phi = \text{`}\parallel\text{'})$$

Theorem 8. *Comb is a plausible TSS.*

Proof. This result follows from Theorem 4 since each component clock is plausible. □

As an illustration, if the clocks REV and KLA used to timestamp the execution presented in Figs. 2 and 6 were combined to timestamp the same execution history, the number of errors is reduced to 38 out of 400 pairs of events (ρ = 0.095). This result is consistent with Theorem 4.

5.4 Discussion

This section describes briefly some refinements to the implementations of REV, KLA and $Comb$ clocks. The purpose of these changes is either to improve the ordering accuracy of the clocks or to reduce the overhead that such clocks impose.

Many mappings between sites and entries of the vector are possible under REV. Given two different mappings f

[3] As Sect. 5.4 suggests, this timestamp can be optimized by including just one copy of the site identification.

and g, it is likely that they induce different results for REV. An interesting option is to divide the R entries of the vector in two or more parts, each with a different mapping and then, using Definition 6, their individual results are combined. If the mappings are chosen in such a way that their results about concurrent events are as disjoint as possible, their combination tends to be more accurate. The selection of the mapping of sites to entries could benefit from previous knowledge about the communication patterns among sites. In particular, if two sites communicate frequently among themselves, most of their events will be ordered and therefore better results can be obtained by making them share the same entry in the vector, or equivalently, if two sites do not interact very frequently, it is better if they use different entries in the vector.

From rule **KL2**, we can notice that when a site receives a message timestamped by KLA, entry K-1 of this timestamp is not used to update the local clock of the receiving site. Therefore, there is no need to send this entry with each message in the system. Furthermore, if **V** is the vector of integers of a timestamp generated by KLA, we have that for any k $<$ K-1 it holds that if **V**[k] > 0, then **V**[k] $>$ **V**[k+1], or if **V**[k] = 0, then **V**[k] = **V**[k + 1], i.e. if there are any entries in **V** whose value is zero, they will be together in the rightmost positions of **V**. This means that there is no need to store, compare and send all the K entries of a KLA clock: we only have to worry about the nonzero entries since the other entries can be filled out with zeroes when needed. Under this scheme, variable-length timestamps can help in reducing the overhead of KLA (the length of a timestamp is the number of non-zero entries in its corresponding vector).

In order to implement variable-length timestamps, we now represent the logical time of site i as $\langle i, \mathbf{Q}, \mathbf{V}_i \rangle$, where i and **Q** are integers, and $\mathbf{V}_i$ is a vector of integers with **Q** entries. Notice that $1 \leq \mathbf{Q} \leq$ K. When local events are generated one after another at site i, its component **Q** is not altered because only $\mathbf{V}_i$[0], i.e. the Lamport clock, is updated. **Q** is incremented and the length of $\mathbf{V}_i$ grows every time that a message is received, unless that **Q** is already greater than the length of the timestamp received in the message. This growth stops when **Q** is equal to K.

A variation of the previous scheme allows **Q** to be incremented even if $\mathbf{V}_i$ has reached its maximum physical length of K entries. In this case, **Q** becomes the *logical length* of $\mathbf{V}_i$, while the actual length of $\mathbf{V}_i$ is min(K, **Q**). If $\mathbf{T}_1$ and $\mathbf{T}_2$ are timestamps generated at two different sites, with logical lengths $\mathbf{Q}_1$ and $\mathbf{Q}_2$, respectively, it holds that:

$$\mathbf{T}_1 \longrightarrow \mathbf{T}_2 \Rightarrow \mathbf{Q}_1 < \mathbf{Q}_2$$

This fact can be used to improve the accuracy of the tests defined in KLA, allowing the correct detection of more pairs of concurrent events.

A combination of clocks, as established by Definition 6, uses the component plausible clocks as black boxes, i.e. we are just interested in their results when two timestamps are compared and not in their internal workings. This is convenient because it only requires that the clocks satisfy the characteristics of a plausible clock. On the other hand, if we have access to implementation details of the clocks, several optimizations may be possible. Duplicate information can be eliminated (e.g., *Comb* keeps two copies of the same site identifier) and relations among internal elements of the combined clocks could reveal new tests to discern more accurately the actual causal ordering between timestamps.

Table 3. Sample groups used

	A	B	C	D
Samples	3	3	3	3
Sites	100	76	77	78
Servers	N.A.	1	2	3
Events	13719	16951	16933	16739
Pairs	62748749	95856633	95652805	93453195

6 Performance evaluation

We now ponder the number of cases for which plausible clocks, such as the ones defined in this paper, fail to report the correct causal relations. We generate random global histories $\boldsymbol{H}$ and use the proposed TSSs to timestamp all the events of these histories. Using the tests associated with each TSS, we determine the causal relation between each pair of events in $\boldsymbol{H} \times \boldsymbol{H}$ and compare these results with the ones produced by vector clocks under the same circumstances.

6.1 Simulation

In the first part of the simulation, we generate a sample history of a distributed system with N sites. In the second part, this history is executed, collecting timestamps and statistics about each one of the TSSs. A sample is a set of N sequences of events.There are 3 types of events: local event, **send** a message and **receive** a message. The samples were of two types: random communication pattern and Client/Server communication pattern. In the first type, any pair of sites can communicate with each other. The probability of site i sending a message to site j is the same $\forall\, i, j <$ N. In the second type of sample, the sites are divided into *clients* and *servers*. Client sites can communicate only with server sites in a request/reply fashion, where the client first sends a message to a server and the next event is a **receive** from this server. We assume that servers don't send unsolicited messages (e.g. callbacks) to the clients. Servers are free to communicate among themselves in a random fashion, but for each message that they receive from a client, the next event must be a **send** to this client.

The simulation executes the particular history of each site, sending and receiving messages and keeping the timestamps assigned to each event by each one of the TSSs that we are evaluating, together with standard vector clocks. After that, using the tests defined by each TSS, we decide the causal relationships between all the ordered pairs from the set $\boldsymbol{H} \times \boldsymbol{H}$ and compute the parameter ρ.

6.2 Results

A total of 347,711,382 pairs of events distributed in 4 groups (A, B, C and D) of 3 samples each, were used to evaluate the proposed TSSs. Group A exhibits a random communication

Table 4. Values of ρ for the evaluated TSSs

	A	B	C	D	Aver.
REV(R=3)	0.446	0.141	0.150	0.153	0.202
KLA (K=3)	0.521	0.076	0.137	0.166	0.197
Comb (R=3, K=3)	0.388	0.071	0.112	0.127	0.154

pattern. The other 3 groups have a Client/Server communication pattern with 1, 2 and 3 servers respectively. Table 3 shows statistics for each group of samples.

Table 4 presents the rate of errors (ρ) that was obtained for each of the samples when timestamped by each TSS described in Sect. 5. The final column of the table shows weighted averages of this parameter. As it was predicted by Theorem 4, $Comb$ consistently produces the minimum values of ρ for exactly the same samples. The evaluations indicate that $Comb$ has excellent performance when the communication pattern is Client/Server. In particular, the best results are obtained for Group B (1 server and 75 clients), where this TSS, with just an overhead of 7 elements, correctly determined the causal relation between 92.9% of the 95,856,633 pairs of events considered. If we take into account all the 347,711,382 pairs of events, we find that $Comb$ is correct in 84.6% of the cases. In the random case where errors are high, $Comb$ improves accuracy over 13% compared to either REV or KLA. Better results can be expected for higher values of R and K.

6.3 Effects of the values of R and K

By decreasing the number of components in the vector clocks, we are improving the efficiency of clock operations but decreasing the accuracy with which they detect orderings between events. The results of Table 4 indicate that a small number of entries (in a Client/Server communication pattern) correctly capture a large number of the causal relations between events. We are interested in investigating the behavior of the described plausible clocks when the number of entries of these clocks is increased. Ideally, it should be possible to capture most orderings with small values of R and K.

With a simulation study (7078 events and 50,098,084 pairs of events), we relate the ordering accuracy of REV and KLA with the size chosen for them. The sample has a Client/Server communication pattern and simulates a distributed system with 99 client sites, 1 server site, an average of 60 events per site and an average of 35 messages from each site. Figure 7 plots the obtained values of ρ for this sample when R and K are varied from 2 to 99 entries.

In general, the rate of errors reduces when the size of the vector is increased. The REV curve presents a fast reduction of ρ during the initial increase in R, e.g. it decreases from 0.167 to 0.097 when R is increased from 2 to 15 entries. However, after that point the pace of reduction of ρ slows down. In order to move ρ down from 0.097 to 0.02, R must be increased from 15 to 77. Obviously, a value of R=100 makes REV equivalent to standard vector clocks and therefore ρ would be 0.0. On several occasions, an increase in the number of entries used by REV actually increases the value of ρ slightly. However, these increases are very small and of local nature and overall the tendency is towards a reduction of ρ as the size of the vector is increased.

The KLA curve shows an excellent start, with a rate of errors of 0.156 for $2LA$ which is reduced to 0.083 for $3LA$ and to 0.079 for $5LA$. The minimum value of ρ is 0.078 with 15 entries, from that point on there are no improvements when more entries are added to the clock. In fact even with as many entries as sites, the rate of errors never gets to zero. It is interesting to consider the problem of how to distribute a given number of entries between REV and KLA. This simulation experiment demonstrates that with modest size plausible clocks, a large number of orderings can be captured correctly.

Figure 7 may suggest a connection between the performance of a plausible clock and the number of sites in the system (i.e., the accuracy decreases when N increases). Even when this behavior would be expected, our preliminary results seem to indicate that the accuracy of plausible clocks may be more sensitive to other factors, such as: communications patterns, size of the global history, level of concurrency in the system, frequency of communications, etc. In order to address this interesting question, it is necessary that a more detailed study be developed which considers the effects of many factors in the performance of plausible clocks.

7 Some applications of plausible clocks

Plausible clocks strive to provide a high level of accuracy in ordering events in a distributed system but they do not guarantee that concurrent events are not ordered. Thus, such clocks are useful for any application where imposing orderings on some pairs of concurrent events has no effect on the correctness of the application. Notice that given the imperfection of plausible clocks, some applications could incur inefficiencies from time to time. However, such inefficiencies due to unnecessary orderings of concurrent events will not induce wrong results, and if the frequency of these orderings is relatively low, the loss in performance is compensated by the potential for scalability and the savings in communications overhead, storage costs and timestamp processing. We describe some applications where plausible clocks can be used.

7.1 Concurrency measures

A concurrency measure is a metric that ponders how concurrent a computation is. It takes into account the causal relationships between the events in a given finite history and how many events can be executing concurrently at a given instant. Let us consider two concurrency measures proposed by Charron-Bost [5], namely ω and m. The key element to compute both of them is the ability to detect concurrent pairs of events, which can be done using vector clocks. Plausible clocks fail in detecting some of the pairs of events that are concurrent, but can be used to obtain approximated values for these metrics.

The first concurrency measure is the ratio of pairs of concurrent events to the total number of pairs of events occurring at different sites, i.e.:

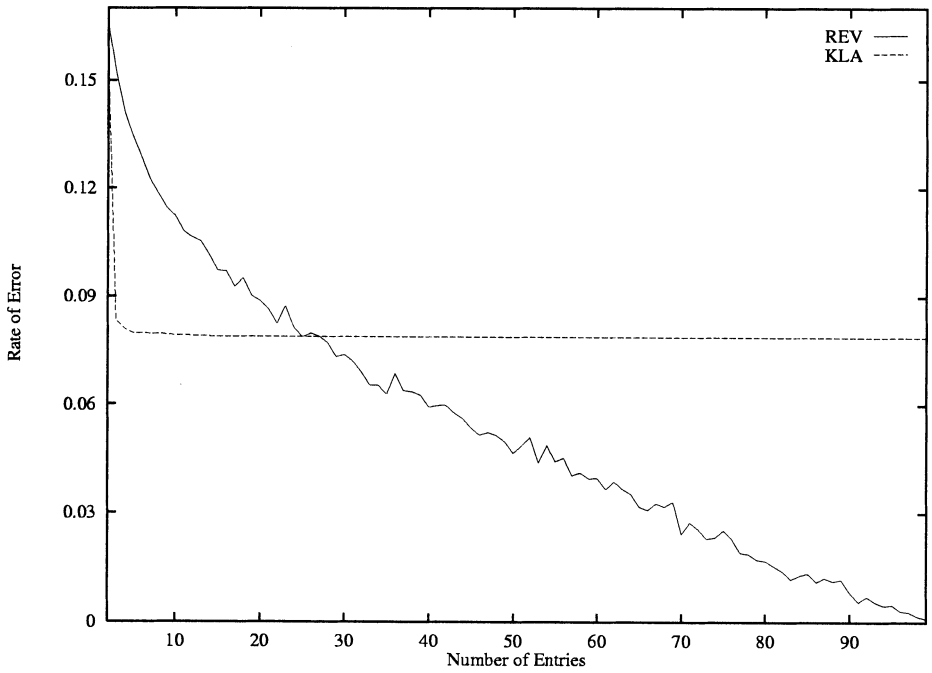

Fig. 7. Varying the values of R and K

$$\omega(\boldsymbol{H}) = \frac{|\{\{\mathbf{a},\mathbf{b}\} : \mathbf{a},\mathbf{b} \in \boldsymbol{H} \wedge \mathbf{a} \parallel \mathbf{b}\}|}{|\{\{\mathbf{a},\mathbf{b}\} : (\mathbf{a} \in \boldsymbol{H}_i) \wedge (\mathbf{b} \in \boldsymbol{H}_j) \wedge (i \neq j)\}|}$$

This measure can be accurately computed by assigning vector clocks to all the events in $\boldsymbol{H}$. Now, if we use a plausible clock P instead of vector clocks, we obtain the quantity:

$$\omega_P(\boldsymbol{H}) = \frac{|\{\{\mathbf{a},\mathbf{b}\} : \mathbf{a},\mathbf{b} \in \boldsymbol{H} \wedge \mathbf{a} \overset{P}{\parallel} \mathbf{b}\}|}{|\{\{\mathbf{a},\mathbf{b}\} : (\mathbf{a} \in \boldsymbol{H}_i) \wedge (\mathbf{b} \in \boldsymbol{H}_j) \wedge (i \neq j)\}|}$$

Since the number of pairs of concurrent events detected by P is less or equal than the actual number of pairs of concurrent events, we have that $\omega_P(\boldsymbol{H}) \leq \omega(\boldsymbol{H})$. The difference $\omega(\boldsymbol{H}) - \omega_P(\boldsymbol{H})$ is a function of $\rho(P)$, i.e. the better the plausible clock P is, the closer $\omega_P(\boldsymbol{H})$ is to $\omega(\boldsymbol{H})$.

Let $\boldsymbol{G} \subseteq \boldsymbol{H}$ be a set of events formed by taking a prefix from each one of the N local histories. $\boldsymbol{G}$ is a *cut* of history $\boldsymbol{H}$. As defined by Mattern [19], we say that $\boldsymbol{G}$ is a *consistent cut*, if it has the property:

$$\forall\, \mathbf{a},\mathbf{b} \in \boldsymbol{H} : (\mathbf{b} \in \boldsymbol{G}) \wedge (\mathbf{a} \longrightarrow \mathbf{b}) \Rightarrow \mathbf{a} \in \boldsymbol{G}$$

If we insert one artificial event, called *cut event*, after each one of the N prefixes taken from the local histories and these N cut events are concurrent among themselves, then $\boldsymbol{G}$ is a consistent cut [19].

The concurrency measure m is defined as:

$$m(\boldsymbol{H}) = \frac{\mu - \mu^S}{\mu^C - \mu^S}$$

Here μ is the number of consistent cuts in $\boldsymbol{H}$, μ^C is the number of consistent cuts if there was no communication between any of the sites (i.e. all the **send** and **receive** events are considered just local events without exchange of messages) and μ^S is the number of consistent cuts if the execution were totally sequential, e.g., all the events of site 0 are followed by all the events of site 1 and so on. Notice that all we need to compute the values μ^C and μ^S is the number of events executed by each site. The quantity m is demonstrated to be a better measure than ω, because when it considers the number of consistent cuts, it is taking into account larger collections of events that are concurrent among themselves (i.e. longer antichains [5]).

Since plausible clocks may fail to detect that two events are concurrent, the test of the simultaneous concurrency of N cut events as described above is not convenient for finding consistent cuts with plausible clocks. However, we can consider a slightly different approach. Given a cut $\boldsymbol{G}$, we say that $\mathbf{last}_i \in \boldsymbol{H}_i$ is the latest event of $\boldsymbol{H}_i$ that is included in $\boldsymbol{G}$, and $\mathbf{next}_i \in \boldsymbol{H}_i$ is the earliest event of $\boldsymbol{H}_i$ that is not included in $\boldsymbol{G}$ (i.e., $\mathbf{last}_i$ is the last event of the prefix taken from $\boldsymbol{H}_i$ and $\mathbf{next}_i$ is the first event of $\boldsymbol{H}_i$ after such prefix). $\boldsymbol{G}$ is a consistent cut if no $\mathbf{next}_i$ $(0 \leq i \leq \mathrm{N}-1)$ is causally before any $\mathbf{last}_j$ $(0 \leq j \leq \mathrm{N}-1)$[4]. If this test is implemented with plausible clocks, it will reject any cut that is not consistent. Furthermore, it is less prone to missing consistent cuts compared to the test of concurrent cut events. For instance, the distributed computation presented in Fig. 1 has 636 consistent cuts and using the plausible clock REV (R=2) we find 476 of them. Nevertheless, there may be consistent cuts where a certain $\mathbf{next}_i$ is concurrent with a given $\mathbf{last}_j$ (which does not affect the consistency of the cut) but the plausible clock reports that $\mathbf{next}_i$ is causally before $\mathbf{last}_j$ and then the cut is classified as not consistent. Given a plausible clock P, we say that $\boldsymbol{G}_P \subseteq \boldsymbol{H}$ is a *plausible consistent cut under P* (or just *plausible cut* for short), if it has the property:

$$\forall\, \mathbf{a},\mathbf{b} \in \boldsymbol{H} : (\mathbf{b} \in \boldsymbol{G}_P) \wedge (\mathbf{a} \overset{P}{\longrightarrow} \mathbf{b}) \Rightarrow \mathbf{a} \in \boldsymbol{G}_P$$

It can be easily proved that every plausible cut is a consistent cut [26]. However, it is possible that a number of consistent cuts are not recognized as plausible cuts. We define μ^P as

[4] Alternatively, if V is a vector clock, $\boldsymbol{G}$ is consistent iff $V(\mathbf{last}_j)[i] \leq V(\mathbf{last}_i)[i]$ for $0 \leq j \leq \mathrm{N}-1$ and $0 \leq i \leq \mathrm{N}-1$

the number of plausible cuts in a given history and compute the quantity:

$$m_P(\boldsymbol{H}) = \frac{\mu^P - \mu^S}{\mu^C - \mu^S}$$

Since $\mu^P \leq \mu$ we have that $m_P(\boldsymbol{H}) \leq m(\boldsymbol{H})$, and that the difference $m(\boldsymbol{H}) - m_P(\boldsymbol{H})$ is a function of $\rho(P)$, i.e. the better the plausible clock P is, the closer $m_P(\boldsymbol{H})$ is to $m(\boldsymbol{H})$.

7.2 Distributed resource sharing

Lamport [18] introduced a fair resource sharing problem and solved it using a logical clock. This algorithm provides mutually exclusive access to a resource and guarantees that if the request of process p_{i} is causally ordered before the request of p_{j}, p_{i} is allowed to access the resource first. Timestamps read from logical clocks maintained at the processes are used to order their requests. The algorithm is completely distributed and requires a large number of messages (3N for N processes) for each access to the resource. More efficient algorithms for distributed mutual exclusion exist, however we explore a simple server based solution to the problem. Such solutions are natural (e.g. the site having the resource implements the synchronization server) and have been used widely in distributed shared memory systems where synchronization operations are used to coordinate access to shared data.

Consider a set of processes $p_1, p_2, \ldots, p_{\mathrm{N}}$ that compete for a shared resource **R** and access it in a mutually exclusive fashion. Access to **R** is controlled by a synchronization server process p_{S}. Thus, a process p_{i} ($1 \leq \mathrm{i} \leq \mathrm{N}$) must send a request message to p_{S} and wait for it to grant the resource before p_{i} can access it. We assume that in addition to sharing **R**, the processes $p_1, p_2, \ldots, p_{\mathrm{N}}$ also communicate among themselves to meet their cooperation and coordination needs. These additional messages can create causal orderings among requests of different processes that are sent to the server p_{S}. We want such causal orderings to be respected when p_{S} grants access to the resource to various processes.

One straightforward solution to this problem, based on scalar Lamport clocks, timestamps requests with clock values and p_{S} grants the requests in timestamp order. However, before a request is granted, p_{S} must ascertain that no request with a lower timestamp is in transit or will be received at a later time. This can be achieved if p_{S} sends a message to all processes and the processes respond with an acknowledgment. When communication channels are FIFO, this will ensure the property that no causally preceding requests can be received by p_{S} after it grants a request from some process. (This is similar to the request and acknowledgment messages in the original algorithm presented in [18]).

The performance of the algorithm can be improved by reducing the number of messages as follows. Process p_{S} needs to send a message to a process p_{i} and receive a response from it only to ensure that a future request from p_{i} will not have a timestamp smaller than the request that it wants to grant next. If p_{S} stores the largest timestamp received from each process and if the timestamp for p_{i} is concurrent with, greater than or equal to the timestamp of the request, p_{S} does not need to send a message to p_{i}. This is because without such a message and its response, p_{S} knows that any future request of p_{i} will not have a timestamp smaller than the timestamp of the request being considered.

The average message cost of accessing the resource in this algorithm will depend on the communication pattern among the processes and the ordering accuracy of the clock system that is used to timestamp the requests. It should be noted, however, that as long as the weak clock condition is met by the clock system, the correctness of the algorithm is guaranteed. If a vector clock is used instead of the scalar logical clock, on the average, the number of unnecessary messages sent by p_{S} to p_{i} can be decreased. This is because p_{S} need not send a message to p_{i} when its timestamp stored at p_{S} is concurrent with the timestamp of the request under consideration which is from p_{j}. Since the event **a** that corresponds to the last communication between p_{i} and p_{S}, and the event corresponding to p_{j}'s request could be concurrent, and vector clocks identify such events accurately, no unnecessary messages will be sent in this case. (A scalar clock may order event **a** before p_{j}'s request which will result in p_{S} sending a message to p_{i}).

Although vector clocks reduce unnecessary messages, they do not eliminate them completely. Process p_{i} could have sent a message to p_{j} after event **a**. In this case, p_{S} must ascertain that no request from p_{i} is timestamped between the clock value it stores and the timestamp of p_{j}'s request. In this algorithm, the performance improvements are possible because of message savings in cases when the event **a** of p_{i} that corresponds to its last communication with p_{S} is concurrent with the request of p_{j} that the server wants to grant next. Plausible clocks can identify such concurrent events more accurately than scalar clocks. On the other hand, if these concurrent events appear ordered from the plausible clock timestamps, p_{S} may have to send an unnecessary message but the correctness of the algorithm is not compromised.

7.3 Object consistency

Distributed systems are increasingly being used to support sharing among widely distributed users. If the shared information is encapsulated in objects, replication and caching of object state is necessary to provide high availability and performance, and to deal with problems such as disconnection that arise in mobile environments. Both replication and caching create problems of consistency among multiple copies of related objects.

A number of consistency criteria have been developed that meet the sharing needs of many types of applications. In this section, we explore one criterion, called causal consistency (**CC**). **CC** has been shown to be sufficient for applications that support asynchronous sharing among distributed users. It has been explored both in message passing systems [4] and in shared memory and object systems [1, 16, 17]. **CC** ensures that values read at a site are consistent with the causality order [18]. This order is established because of local order between operations at a process, and a read-from order that orders operation **w** before operation **r** when **r** reads the value written by **w**. For example, in Fig. 8, once p_2 reads value v_3 of y, its future read of x cannot return v_1. This is because, according to causal order, the second read

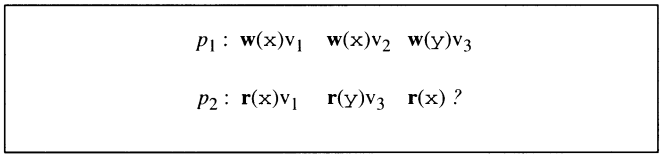

Fig. 8. Last read of p_2 can not be v_1

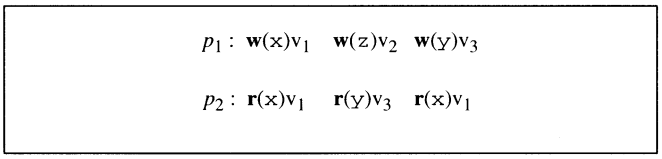

Fig. 9. Causally consistent execution

of x by p_2 occurs after the operation **w**(x)v_2 of p_1. Since this operation overwrites the value v_1 of x, v_1 must not be read by p_2 after its read of y.

In an implementation of **CC** described by Ahamad, John, et al. [2, 16], a client can access objects that are in its cache freely. However, when a new object value is introduced into the site cache, it is necessary to ensure that existing values do not become causally overwritten as a result of reading the new value that is added to the site cache (such a read can create new causal orderings). For example, in the execution shown in Fig. 8, once the value v_3 of y arrives at p_2, the value v_1 for x can no longer be accessed because it has been overwritten.

There are two options for handling new object values when they arrive at a site. The site can either wait until all causally preceding values are received or it can invalidate existing values that it suspects are causally overwritten. The latter is preferred when client access patterns are dynamic and it is undesirable to send values of updates to all sites that may have copies of the objects. Therefore, we consider the implementation in which **CC** is maintained by invalidating cached copies that may be potentially overwritten according to causality.

The problem of detecting what values are causally overwritten when a new value of object x is received at a site is solved by Ahamad, John, et al. [2, 16] by associating timestamps read from a logical clock with copies of objects. These timestamps capture the logical time at which the object copy was produced by a write operation. If the timestamp of object y that is added to a site cache is T, all the existing objects x at the site are invalidated if their associated timestamps are less than T. This is done because the cached objects may potentially be overwritten by more recent operations that occurred before time T. On the other hand, if the timestamps of x and y are concurrent, the two copies can coexist without violating consistency. If Lamport clocks are used to determine the ordering between operations that produced the object copies, objects that were produced by concurrent operations may appear to be generated by ordered operations because these clocks can order concurrent events. As a result, objects can be unnecessarily removed from the site cache.

Such unnecessary removals can be avoided if the timestamps associated with object copies are derived from more precise clocks. This could result in improved performance by not removing consistent object copies and avoiding communication on accessing such objects in the future. Once again, vector clocks can precisely order events of a distributed system. Concurrent events are detected to be so by these clocks, therefore object copies produced by concurrent operations could co-exist in a cache and it is not necessary to remove such copies from a site cache to preserve causal consistency.

When vector clocks are used, a causally overwritten value will always have a lower timestamp than the timestamp of an incoming object copy but the reverse may not be true. For example, in Figs. 8 and 9, p_2 will receive the same vector clock with the copy of y. However, in Fig. 8 the value of x should be invalidated whereas this is unnecessary in the execution shown on Fig. 9. Since p_2 cannot distinguish between the two situations just by looking at the vector timestamps, in both cases the existing value of x is invalidated. Therefore, even a vector clock based system can suffer from unnecessary invalidations.

The ability to accurately detect if the two writes that produced the values of x and y are concurrent or not, only impacts the performance of the consistency algorithm by reducing the number of unnecessary invalidations. Thus, plausible clocks can be used instead of vector clocks in such implementations, avoiding the high costs of vector clocks. At the same time, more concurrent operations are detected with plausible clocks than with a Lamport Clock, which implies a reduction in the number of unnecessary invalidations of objects and therefore improved performance of the system.

As an example, Table 5 presents the results from a simple simulation of a set of processes that share causally consistent objects, using the techniques originally described by Kordale and Ahamad [17], and later refined by Torres-Rojas, Ahamad and Raynal [28]. **CC** is ensured by invalidating cached copies that are potentially overwritten as determined by timestamps read from a logical clock. When an object copy is updated by a process, a message with the new value of the object is sent to the server of the system. The cache misses are reported to the server as well, which in turn responds by sending the last value of the requested object known to the server. It is at this point that object copies that may have been potentially overwritten are invalidated. Notice that these invalidations do not generate messages to any other processes, i.e. invalidations are local operations. However, if there are object copies in the local cache whose timestamps are greater than the timestamp of the received object, then the received object must be validated by sending messages to the server and other client processes, because it may have been overwritten.

A total of 49 client processes and 1 server process, executing 835 operations (336 updates) over 4 objects were simulated[5]. The first row of Table 5 shows the statistics collected when 50-entries vector clocks were used to establish the causal relationships between the events that produced the object values residing in a particular cache at a given time.

[5] For the purposes of this simulation, we are only interested in the number of messages and the size of the required timestamps

Table 5. Causal consistency with vector clocks and plausible clocks

	Updates		*Cache Misses*		*Invalidations*		*Validations*		*Messages*
Clock Mechanism	Total	Mess.	Total	Mess.	Total	Mess.	Total	Mess.	Total
Vector Clocks	336	336	103	206	6	0	3	5	547
Plausible Clocks	336	336	109	218	34	0	11	20	574

Similarly, the second row shows the same statistics when the same history was executed using a 6 element plausible clock $Comb$. The number of cache misses, invalidations, validations and their associated messages obtained with plausible clocks are not much bigger than the corresponding numbers for vector clocks. On the other hand, each one of the 547 messages generated by the execution with vector clocks must carry a 50 entry vector, i.e. an overhead of 27350 integers, while each one of the 574 messages generated by the execution with plausible clocks must include a 6 entry vector, i.e. an overhead of just 3444 integers. Both algorithms produce a causally consistent execution.

8 Related work

Fundamental concepts for ordering in distributed systems such as logical clocks, partial orderings of events and the "happens before" relation, were all presented in the seminal paper by Lamport [18]. Fidge [10, 11, 12] and Mattern [19] independently proposed the technique of vector clocks that permits a complete characterization of causality. Charron-Bost [6] proved that this characterization can only be done with vector clocks where there is one entry for each site in the distributed system. Cheriton and Skeen [7] mention problems in scalability that are incurred by systems where causally and totally ordered communications are implemented using vector clocks. Reviews of diverse techniques for representation of logical time and its applications are presented by Raynal [21], Raynal and Singhal [22] and Schwarz and Mattern [25]. A compilation of some of the most significant papers on logical time and global states in distributed systems was prepared by Yang and Marsland [30].

There have been many efforts to reduce the overhead imposed by vector clocks. We review these briefly here. Haban and Weigel [14] use vector clocks as part of their mechanism for defining global breakpoints in a distributed system. In order to save space, they allow processes to share an entry in the vector clock when they are executed on the same physical processor. This is similar to the basic idea of REV. Notice, however, that REV permits different processes to share the same entry even if they are executing at different sites. Meldal, Sankar and Vera [20] assume a communication network that is static and reasonably sparse. In that case, site i does not need to keep entries in its vector clock to represent sites that don't communicate with i. If eventually a new communication link is established between two sites, the size of the vectors is adjusted appropriately. Thus, in the worst of cases, their clocks can grow up to the size of standard vector clocks. Plausible clocks make no assumption about the connectivity of the sites in the system.

A technique to reduce the size of timestamps appended to messages is proposed by Singhal and Kshemkalyani [24]. It is based on the observation that a given site tends to interact frequently with only a small set of other sites and that timestamps assigned to two consecutive events by a site differ in just a few entries. This technique reduces communications overhead but, as it is mentioned by Meldal et al. [20] and Schwarz and Mattern [25], since the information in timestamps is compressed, there are cases when the causality relationship between different messages sent concurrently to the same site can not be correctly established. These clocks, similarly to plausible clocks, may order some concurrent events. Notice, however, that even though the size of timestamps carried by messages is reduced, the size of several data structures stored at each site depends on the number of sites in the distributed system. A different technique is described by Fowler and Zwaenepoel [13], where each site maintains a vector $\mathbf{V}_i$ with N entries. Site i tags the messages that it sends with just $\mathbf{V}_i[i]$, eventually this value will update entry i of the receiver's vector. This technique fails to represent transitive dependencies and is more useful for applications where the causal dependencies are determined off-line [22].

Ahuja, Carlson and Gahlot [3] propose a model for distributed systems where the notion of each process as a sequence of events is eliminated, i.e. the complete system is considered a partially ordered set of events. This allows the presence of not only interprocess concurrency but also of intra-process concurrency. In order to detect these new sources of concurrency, a clock with much more complexity than vector clocks is required. The authors explain a mechanism to trade cost and concurrency identification, that at a minimum reduces their clock to a standard vector clock, but sacrifices their ability to detect intra-process concurrency. Notice that if a unique identification is available for each one of the threads of sequential execution inside a process, plausible clocks could be used to detect a fraction of both inter-process and intra-process concurrency.

Diehl and Jard [9] presented the timestamping technique known as "interval clocks" that obtains better results than a Lamport clock in ordering events and has a similar cost. They recognize the importance of approximating the causality relation with a realistic overhead. In that sense, our work with plausible clocks can be considered a generalization and extension of such ideas. Valot [29] studies the trade-offs (space vs. accuracy) that Lamport clocks, interval clocks and vector clocks offer when used as timestamping mechanisms. She concludes that, according to the knowledge that we have of the nature of the execution, it may be possible to obtain an acceptable level of accuracy without using excessive space in timestamps; the parameter accuracy defined

in Valot's paper can be obtained from the parameter ρ (rate of errors) presented in Definition 7.

All the logical clocks mentioned in this paper are based on structures built with integer numbers. This fact poses the practical issue of how to deal with the capacity limits of integer representation. Several solutions to this problem have been proposed, e.g. [15] and [23]. Notice that this issue can be addressed orthogonally to the chosen representation for logical time, being this Lamport clocks, vector clocks or plausible clocks.

9 Conclusions

There is an isomorphism between vector clocks and the causality relation of events in a Distributed System. Therefore, vector clocks are useful in understanding the behavior of distributed systems. However, they have the major disadvantage of not being constant in size: their implementation requires the presence of an entry for each one of the N sites in the distributed system. Charron-Bost's results [6] discourage any attempt to define some kind of clock that, while constant in size, completely captures the causality relation.

We propose a class of logical clocks called plausible clocks that can be implemented with a constant number of components and yet they provide, under certain circumstances, ordering accuracy close to vector clocks. We develop rules to combine known plausible clocks to produce more accurate clocks.

Several implementations of constant size plausible clocks are presented. REV is a variant of vector clocks where R-entries vectors are used; since R $<$ N several entries are shared by more than one site of the distributed system and therefore a mapping between sites and entries in the vector must be defined. KLA is an extension of Lamport clocks where each site keeps a standard Lamport clock together with a collection of the maximum timestamp of any message received by itself and by the K-2 previous sites that directly or indirectly have had communications with this site. $Comb$ is a combination of REV and KLA, and as such can be proved to be at least as good (and possibly better than) as any of its components. These implementations were evaluated using a simulation model and we found that even with a small number of components in REV and KLA, ordering among a large number of events can be detected accurately when the communication pattern is Client/Server. We also presented examples of applications that could benefit from plausible clocks.

We claim that any constant size clock must be plausible in order to be useful, but evidently, there are many other possible implementations of plausible clocks that would be interesting to consider. We have to evaluate the effects that diverse factors have on the performance of plausible clocks (e.g., number of sites in the system, communications patterns, size of the global history, level of concurrency in the system, frequency of communications, etc.). Also, we will explore how to distribute a given number of entries of a vector between REV and KLA clocks.

Acknowledgments. We were greatly helped by the comments and suggestions of the anonymous reviewers.

References

1. M. Ahamad, M. Raynal, G. Thiakime: An adaptive architecture for causally consistent services. Proc. of 18th International Conference on Distributed Computing Systems, ICDCS'98, Amsterdam. 1998
2. M. Ahamad, P. Hutto, R. John: Implementing and Programming Causal Distributed Shared Memory, Proc. of 11th International Conference on Distributed Computing Systems, 1991
3. M. Ahuja, T. Carlson, A. Gahlot: Passive-space and Time View: Vector Clocks for Achieving Higher Performance, Program Correction, and Distributed Computing, IEEE Transactions on Software Engineering, Vol 19, No. 9, September 1993
4. K. Birman, A. Schiper, P. Stephenson: Lightweight Causal and Atomic Group Multicast, ACM Trans Comput Syst 9(3) 272–314 (1991)
5. B. Charron-Bost: Combinatorics and Geometry of Consistent Cuts: Application to Concurrency Theory, Proc. Int. Workshop on Parallel and Distributed Algorithms, Nice, France, pp 45–56, 1989
6. B. Charron-Bost: Concerning the size of logical clocks in Distributed Systems, Inform Process Lett 39: 11–16 (1991)
7. D.R. Cheriton, D. Skeen: Understanding the Limitations of Causally and Totally Ordered Communications, Operating Systems Review, Vol 27, No. 5, December 1993
8. B.A. Davey, H.A. Priestley: Introduction to Lattices and Order, Cambridge University Press, 1990
9. C. Diehl, C. Jard: Interval approximations of message causality in distributed executions, Proc. Symposium on Theoretical Aspects of Computer Science, Cachan, France, pp 363–374, February 1992
10. C.J. Fidge: Timestamps in message-passing systems that preserve the partial ordering, Proc. 11th Australian Computer Science Conference, University of Queensland, pp 55–66, 1988
11. C.J. Fidge: Logical Time in Distributed Computing Systems, Computer 24(8) 28–33 (1991)
12. C.J. Fidge: Fundamentals of Distributed System Observation, IEEE Software, Vol 13, No. 6, November 1996
13. J. Fowler, W. Zwaenepoel: Causal distributed breakpoints, Proc. of 10th Int'l. Conf. on Distributed Computing Systems, pp 134–141, 1990
14. D. Haban, W. Weigel: Global Events and Global Breakpoints in Distributed Systems, Proc. 21st Hawaii International Conference on Systems Sciences, January 1988
15. A. Israeli, M. Li: Bounded Time-stamps, Proceedings of Twenty-eighth Annual Symposium on Foundations of Computer Science, pp 371–382, 1987
16. R. John, M. Ahamad: Evaluation of Causal Distributed Shared Memory for Data-racefree Programs, Technical Report, College of Computing, Georgia Institute of Technology, 1991
17. R. Kordale, M. Ahamad: A Scalable Technique for Implementing Multiple Consistency Levels for Distributed Objects, Proceedings of the 16th International Conference on Distributed Computing Systems, May 1996
18. L. Lamport: Time, clocks and the ordering of events in a Distributed System, Communications of the ACM, Vol 21, pp 558–564, July 1978
19. F. Mattern: Virtual Time and Global States in Distributed Systems, Conf. (Cosnard et al. (eds)) Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, North-Holland: Elsevier, pp 215–226, 1988
20. S. Meldal, S. Sankar, J. Vera: Exploiting locality in maintaining potential causality. Proc. 10th Annual ACM Symposium on Principles of Distributed Computing, Montreal, Canada, pp 231–239, 1991
21. M. Raynal: About Logical Clocks for Distributed System, Operating Systems Review, Vol. 26, No. 1, January 1992
22. M. Raynal, M. Singhal: Logical Time: Capturing Causality in Distributed Systems, IEEE Computer, Vol. 29, No. 2, 1996
23. A.K. Singh: Bounded Timestamps in Process Networks, Department of Computer Science, University of California at Santa Barbara, June 1992
24. M. Singhal, A. Kshemkalyani: An efficient implementation of vector clocks, Inf. Process Lett. 43: 47–52 (1992)
25. R. Schwarz, F. Mattern: Detecting causal relationships in distributed computations: in search of the holy grail, Distrib Comput 7: 149–174 (1994)

26. F. Torres-Rojas: Efficient Time Representation in Distributed Systems, MSc. Thesis, Georgia Institute of Technology, 1995
27. F. Torres-Rojas, M. Ahamad: Plausible Clocks: Constant Size Logical Clocks for Distributed Systems, Proc. 10th International Workshop on Distributed Algorithms, (WDAG 96). Bologna, Italy, October 1996
28. F. Torres-Rojas, M. Ahamad, M. Raynal, Lifetime Based Consistency Protocols for Distributed Objects, Proc. 12th International Symposium on Distributed Computing, DISC'98, Andros, Greece, September 1998
29. C. Valot: Characterizing the Accuracy of Distributed Timestamps, ACM SIGPLAN Notices, 28 (12), December 1992
30. Z. Yang, T.A. Marsland (eds): Global States and Time in Distributed Systems, IEEE Computer Society Press, 1994

Francisco Torres-Rojas was born in San Jose, Costa Rica. He received a B.Sc. in Computer Science from the Universidad de Costa Rica, and a M.Sc. in Computer Science from the College of Computing at Georgia Tech, where currently he is a Ph.D. candidate. His research interests include operating systems, distributed systems, distributed algorithms, consistency of distributed objects and theoretical issues of parallel and distributed computing.

Mustaque Ahmad is a Professor in the College of Computing at the Georgia Institute of Technology. He received his Ph.D. in Computer Science from the State University of New York at Stony Brook in 1985. His primary research interests are in the areas of distributed operating systems, middleware, and scalable and secure systems. In particular, his current research focus is on scalable consistency protocols for sharing dynamic information, and on system support for interactive applications in the wide-area networking environment.