Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications

Tobias Landes
Institut für Informatik
Technische Universität München
Germany

Abstract

A large number of tasks in distributed systems can be traced down to the fundamental problem of attaining a consistent global view on a distributed computation. This problem is commonly solved by appliance of vector clocks as a means of tracing causal dependencies among the events that characterize a run of the computation. In the paper at hand an extension to the concept of vector clocks is presented and examined that is meant to overcome the vector clocks' great drawback: that the number of processes in the distributed system has to be constant and known in advance. As an appropriate context for these dynamic vector clocks *and their associated algorithms to be integrated into, scalar and vector clocks are analogously reinvestigated.*

Keywords: *distributed system, vector clocks, dynamic, logical time, event order*

1 Introduction

Due to decreasing hardware costs and increasing needs of computing resources, distributed systems steadily gain practical importance. The benefit of distributed systems lies in their ability to carry out a computation in parallel fashion. On the other hand, this makes distributed computations much more complex and difficult to understand than their sequential counterparts.

One fundamental problem presenting itself in manifold tasks, such as system monitoring, breakpointing, or detecting global predicates for debugging purposes, is to retrieve a global view on the computation which represents a specific consistent global state of the system. This fundamental problem results from the fact that in a distributed environment, unlike a non-distributed one, there is no common clock or time base to order the computational events in the sense that one could say event e_i happened before or after e_j. The common approach to overcome this deficiency is to substitute for the global clock a relative "time approximation" based on logical principles like program-inherent causal dependencies between single events. These causal dependencies derive from the program logic in the sense that, first, each process is regarded as a sequence of events ordered inherently by the underlying program and, second, in order for the distributed computation to make sense, the processes do cooperate in some way or the other, which produces additional dependencies between single events of different processes. The most common way of modeling interprocess communication is to have the processes send and receive messages in respective events. The resulting dependencies can intuitively be described by the statement that the event that represents the sending of such a message must happen before the event that stands for the receipt of the same message.

Even more basic than constructing a consistent global view, which it is actually a part of, is the need to enable any receiver, either a process involved in the computation or an independent monitor process, to order the messages it receives according to their logical order. This problem is commonly solved by using *logical clocks* as proposed by Leslie Lamport in his milestone paper [1]. To suit the needs of many real-life applications, which will be discussed later in this paper, Lamport's scalar logical clock has been refined to the more powerful *vector clock*. This commonly used device has yet one serious drawback: The number of processes involved must be constant and has to be known in advance, which is not an acceptable precondition for the many applications based on the dynamic creation and destruction of processes. Therefore, after discussing the features of scalar and vector clocks, the paper at hand introduces an extension to vector clocks making them suitable for applications which dynamically spawn processes. Features and appropriate algorithms for these *dynamic vector clocks* are given and examined in analogy to those of scalar and vector clocks.

The document is organized as follows: Section 2 briefly describes the system model which will be used throughout the document. Section 3 examines in detail the different kinds of logical clocks and their features, dedicating one subsection each to scalar clocks, vector clocks, and the newly proposed dynamic vector clocks. Section 4 summarizes the contents of the paper.

2 System Model

This section introduces a model for distributed systems. For the sake of simplicity, the model shall be as basic as possible, without shared memory or similar sophisticated features, which will be sufficient for the issues discussed in this paper.

In this model, a distributed computation consists of a finite set $P = \{p_1, p_2, \ldots, p_n\}$ of *n processes*. The processes communicate only by sending and receiving *messages*, which are assumed to be delivered reliably and with a finite delay. For many applications it is necessary that the messages be delivered exactly in the order they have been sent, which establishes the notion of *FIFO channels*. Since these are closely bound to logical clocks, they will not be generally assumed, but explicitly mentioned in appropriate places throughout this paper.

Any process p_i consists of a sequence of *events* $E_i = \{e_i^1, e_i^2, \ldots\}$, which are totally ordered by an ordering relation $\rightarrow$ called the *program order*. Each event is atomic on the viewed abstraction level and changes the *state* of the process. E is the set of all events in the system. Of particular interest for considerations regarding the global behaviour of systems with interacting processes are events representing the sending or receiving of a message, i.e. *send* and *receive events*. This is because these events establish synchronization dependencies among the processes and thus extend the local program order to a partial global ordering $\Rightarrow$ of events, which will be formally defined in section 3.

3 Logical Clocks

A fundamental necessity in many distributed applications is that message receiving processes be able to order their incoming messages appropriately, i.e. to string them up and react to them in an order restricted by the system's delivery rule (for example the precise order they have been sent). This need presents itself in the basic operating logic of a distributed computation as well as in higher level applications such as retrieving a global view on the computation that represents a consistent global system state and is used, for example, for system monitoring, breakpointing or debugging purposes.

Since the typical asynchronous distributed system lacks a global clock which could provide a common time base to all processes, Lamport [1] introduced *logical clocks* as a device to substitute for a global real time clock. As the term already suggests, logical clocks are used to order events (or associated entities, like messages) based on their relative logical dependencies rather than on a "time" in the common sense. This means that a statement such as "event e_i happens before event e_j" is to be read as "event e_i must happen before event e_j". Of course there are events for which there is no such dependency, and which, therefore, must (or may) remain unordered. To formally capture all the causal dependencies in a system as modeled in section 2, Lamport [1] defined the "happened before" relation in the following way as a partial order relation, which is the transitive closure of the program order and the natural causal send-receive dependencies:

Definition 3.1. The *"happened before" relation* $\Rightarrow$ is the smallest relation satisfying the following conditions:

- If $e_i^x \rightarrow e_i^y$, then $e_i^x \Rightarrow e_i^y$.
- If e_i^x is a send event and e_j^y is the receive event of the same message, then $e_i^x \Rightarrow e_j^y$.
- If $e_i^x \Rightarrow e_j^y$ and $e_j^y \Rightarrow e_k^z$, then $e_i^x \Rightarrow e_k^z$.

If $e_i^x \Rightarrow e_j^y$, then e_j^y is regarded as being *causally dependent* on e_i^x since it can only be executed if the execution of e_i^x has already been finished. Therefore, e_i^x can also be seen as a *precondition* to e_j^y. Intuitively, this means for example that a message can not be received before it has been sent. If $e_i^x \not\Rightarrow e_j^y$ and $e_j^y \not\Rightarrow e_i^x$, then e_i^x and e_j^y are said to be *concurrent*, and may be executed in parallel since none of them can causally affect the other. Concurrency shall be denoted by $e_i^x \parallel e_j^y$.

It is obvious that any correct logical clock system has to respect the "happened before" relation. Thus Lamport stated the clock condition, which derives from the above relation and will be the base of all logical clock devices:

Definition 3.2. Any correct clock must fulfill the *clock condition*, which is the following:

$$\forall e_i^x, e_j^y \in E : \text{ if } e_i^x \Rightarrow e_j^y \text{ then } C(e_i^x) < C(e_j^y).$$

As a consequence, a correct clock must fulfill two conditions directly derived from the "happened before" relation, one for each of the first two parts of the relation's definition (the third part only states the transitivity of the "happened before" relation and is implicitly covered because $<$ is a transitive order relation):

1. If $e_i^x \rightarrow e_i^y$, then $C(e_i^x) < C(e_i^y)$.
2. If e_i^x is a send event and e_j^y is the receive event of the same message, then $C(e_i^x) < C(e_j^y)$.

Note that these conditions, and the "happened before" relation as of definition 3.1, are system model dependent and in their given form only apply to the basic system model introduced in section 2. Another system model could imply other dependencies among the events, which would also have to be taken into account.

Now a few of the most important goals of using logical clocks shall be briefly described for the sake of motivation and to be referred to in the discussion of the particular clock types in the following sections.

The first significant concept is the *delivery rule*. Delivery rules specify restrictions on the order in which incoming messages are to be delivered to the receiving process for processing. An important and popular example is *FIFO delivery*, which says that a process j shall receive messages sent by process i in the order they were sent, and any other messages (from different processes) in arbitrary order:

$$\text{If } e_i^s \rightarrow e_i^S \text{ then } e_j^r \rightarrow e_j^R,$$

where s/S mark send events and r/R their respectively corresponding receive events. Note that this definition has only the messages between two specific processes ordered, not messages sent by different processes. There

are other definitions of FIFO delivery that demand *all* messages in the system to be delivered in order; this version I will call *strict FIFO delivery*.

Another important delivery rule is *causal delivery*, which is stronger than FIFO delivery as defined above:

$$\text{If } e_i^s \Rightarrow e_k^S \text{ then } e_j^r \rightarrow e_j^R.$$

This expands FIFO delivery to additionally order messages sent by different processes if they are causally related according to the "happened before" relation. Since this delivery rule allows to trace event causality, it is especially important in distributed debugging and system monitoring.

To implement causal delivery a property called *gap detection* is needed. A process receiving two messages with their respective timestamps must be able to decide whether there is another message that belongs in between, but has not yet been received. More formally, given two clock values $C_i(e_i^x) < C_j(e_j^y)$, it has to be determined whether there exists an event e_k^z with $C_i(e_i^x) < C_k(e_k^z) < C_j(e_j^y)$. For a more detailed discussion on this topic see [2].

With these concepts introduced, we can now efficiently discuss different kinds of logical clocks. The oldest and simplest logical clocks were introduced by Lamport as mere scalar values and will be briefly described in the next section.

3.1 Scalar Clocks

Lamport [1] defined a logical clock as a function assigning a simple scalar value to each event in the system:

Definition 3.3. A *logical clock* C is a function which assigns a number $C(e)$ to any event $e \in E$.

To satisfy the conditions stated in the previous section, the clocks C_i, of which every process i maintains its own, are updated with each event e_i^x as follows:

- If e_i^x is not a receive event, then $C_i(e_i^x) = C_i(e_i^{x-1}) + 1$.
- If e_i^x is the receive event of message m, then $C_i(e_i^x) = max\{C_i(e_i^{x-1}), C_j(e_j^y)\} + 1$, where $C_j(e_j^y)$ is the clock value of the send event in process j that sent m to process i. The clock value of any send event is always timestamped on the message sent.

An example illustrating the use of scalar clocks is given in Figure 1. It is easy to verify that these scalar logical clocks satisfy the clock condition. Yet, as handy and simple as they are, they have a serious drawback: They lose causality information, which is acceptable for some applications, but not in general (for example, distributed debugging). One result of the information loss scalar clocks exhibit is the complete lack of the gap detection property, which renders them unusable for applications requiring causal delivery and for applications requiring FIFO delivery, but running on systems which do not provide FIFO channels.

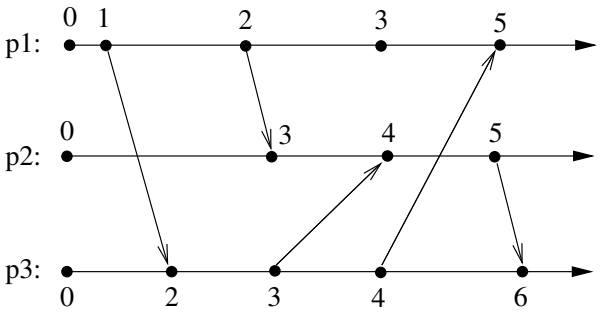

Figure 1: Scalar clocks

Closely related to the gap detection problem, but more important since more general, is the fact that if $C_i(e_i^x) = C_j(e_j^y)$ one can conclude $e_i^x \parallel e_j^y$, whereas the reverse conclusion is not feasible: Given $C_i(e_i^x) < C_j(e_j^y)$ one is still unable to determine whether $e_i^x \Rightarrow e_j^y$ or $e_i^x \parallel e_j^y$, which is often, namely in distributed debugging, a very interesting detail to know. That is, scalar clocks can be used to construct an event ordering which is consistent with causal dependencies, but not to view or analyze the dependencies itself, e.g. to decide whether two events in the ordering could possibly be swapped without losing causal consistency.

3.2 Vector Clocks

Vector clocks are a more powerful extension of Lamport's scalar logical clocks, and commonly used to overcome the deficiencies of the latter. Instead of a single scalar timestamp, vector clocks use several (combined to a vector) in order to capture the causal dependencies among events more comprehensively than the scalar clocks do. Mattern [3] and Fidge [4, 5] have analyzed vector clocks regarding their ability to completely reflect event causality in distributed systems.

As with scalar clocks, each process maintains its own local clock. The clock now is a vector of n scalar values, n being the total number of processes. Each component $C_i(e_i^x)[j]$ holds the latest (scalar) clock value process i knows from process j at the time of event e_i^x.

The update rules for vector clocks are specified as follows:

- If e_i^x is the receive event of message m, then $C_i(e_i^x) = max\{C_i(e_i^{x-1}), C_j(e_j^y)\}$, where $C_j(e_j^y)$ is the clock vector of the send event in process j that sent m to process i and max is the component-wise maximum.
- If e_i^x is any event, including receive events, then the process increments its own scalar clock value: $C_i(e_i^x)[i] = C_i(e_i^{x-1})[i] + 1$.

An example illustrating the use of vector clocks is given in Figure 2.

The great benefit of vector clocks lies in their property not only to fulfill the clock condition as given in definition 3.2, but the following stronger version as well:

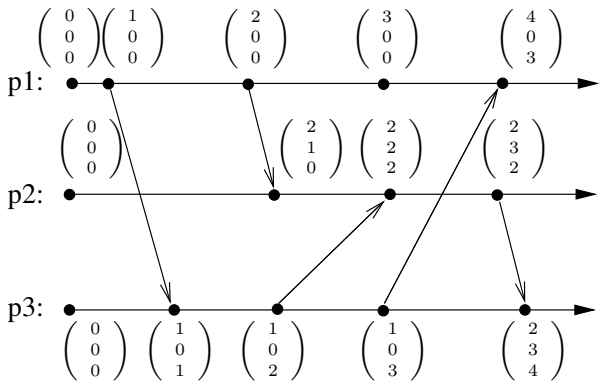

Figure 2: Vector clocks

Definition 3.4. The *strong clock condition* is defined as follows:

$$\forall e_i^x, e_j^y \in E : C(e_i^x) < C(e_j^y) \text{ if and only if } e_i^x \Rightarrow e_j^y.$$

For the vector clocks to work with the strong clock condition, we need to precisely define the $<$ relation for our vectors:

Definition 3.5. The $<$ *relation for vector clocks* shall be defined as follows:

$$\begin{aligned} &C_i(e_i^x) < C_j(e_j^y) \text{ if and only if} \\ &\quad C_i(e_i^x) \neq C_j(e_j^y) \text{ and} \\ &\quad \forall k | 1 \leq k \leq n : C_i(e_i^x)[k] \leq C_j(e_j^y)[k]. \end{aligned}$$

Given these tools, vector clocks allow a simple decision, based on a single scalar comparison, whether one event is causally dependent on another:

$$e_i^x \Rightarrow e_j^y \text{ if and only if } C_i(e_i^x)[i] \leq C_j(e_j^y)[i].$$

Note that this formula is not symmetric. If the formula is applied and yields $e_i^x \not\Rightarrow e_j^y$, it must be applied a second time to decide whether $e_j^y \Rightarrow e_i^x$ or $e_i^x \parallel e_j^y$. This leads directly to the concurrency test:

$$\begin{aligned} &e_i^x \parallel e_j^y \text{ if and only if} \\ &\quad C_j(e_j^y)[i] < C_i(e_i^x)[i] \text{ and} \\ &\quad C_i(e_i^x)[j] < C_j(e_j^y)[j]. \end{aligned}$$

With the $<$ relation applied as defined above and all the update rules obeyed (so the strong clock condition is satisfied) this is equivalent to

$$\begin{aligned} &e_i^x \parallel e_j^y \text{ if and only if} \\ &\quad C_j(e_j^y) \not\leq C_i(e_i^x) \text{ and} \\ &\quad C_i(e_i^x) \not\leq C_j(e_j^y), \end{aligned}$$

which is mentioned here only for the sake of completeness and clarification since in this second form the test requires significantly more scalar comparisons[1].

[1] With n being the number of vector entries (equaling the number of processes), linear $2n$ instead of constant 2 comparisons.

If one of these tests is applied and yields $e_i^x \nparallel e_j^y$, it can be immediately determined, depending on which one of the two conditional terms failed to be true, whether $e_i^x \Rightarrow e_j^y$ or $e_j^y \Rightarrow e_i^x$.

Vector clocks have also the property that they are suited for a restricted form of gap detection, which is, under the additional assumption that processes increment their vector clocks only on events which are notified to a specific monitor process (including all send/receive events), yet sufficient for the implementation of FIFO or causal delivery at the monitor. This property is based on the observation

$$\begin{aligned} &\text{if } \exists k \neq j : C_i(e_i^x)[k] < C_j(e_j^y)[k] \\ &\qquad \text{then } \exists e_k^z : (e_k^z \Rightarrow e_j^y) \wedge \neg(e_k^z \Rightarrow e_i^x). \end{aligned}$$

and is weak in the sense that it lets us decide whether $e_i^x \Rightarrow e_k^z \Rightarrow e_j^y$ only for the special case $k = i$. For a more detailed discussion, see [2].

The great drawback of vector clocks is that they use a fixed index to locate the vector component representing the scalar clock value associated with a given process. Thus two heavily restrictive requirements emerge: First, the number of processes has to be constant and, second, it has to be known in advance. Both of these requirements are not acceptable for inherently dynamic applications, but can be overcome by using the dynamic vector clocks proposed in the next section.

3.3 Dynamic Vector Clocks

The assumption that the number of processes participating in a distributed computation be constant is not quite realistic. Most distributed applications need to be able to grow dynamically according to their individual progress. In this section a straightforward extension of common vector clocks is proposed, which is meant to overcome the vector clocks' deficiencies regarding systems with dynamic process creation.

Since no upper bound on the total number of processes shall be assumed, the clock vector must be able to dynamically grow. Initially it is postulated to have length 1 and thus resemble a scalar local clock of its associated process. In order not to diminish the causality tracing ability of standard vector clocks, a process's vector must continue to gather information about the progress of other processes by merging the information available through incoming messages with its own current clock vector. As with standard vector clocks, the latest of the two available values of each component has to be recorded. Additionally, in a dynamic system, processes previously unknown to the receiver, but already represented in the incoming timestamp, have to be included into the receiver's own vector. The accounting for all possible side effects of the system's dynamics predictably leads to some informational overhead, compared to standard vector clocks, that has to be included in the clock system.

The simplest way to achieve the required vector growing feature would be to include the new process's value

at the position indexed by its process ID (or a function thereof), as is done with standard vector clocks (see section 3.2). However, this practice would lead to significant inefficiencies. Suppose the first message process p_1 receives originates from process p_{337}. As a result, process p_1 would have to fill its clock vector with 335 unnecessary zeros between positions 1 and 337. Needless to say, this would produce a significant overhead not only in memory usage, but especially in network load, because this vector is transmitted along with each and every message process 1 will transmit in the future (and because the waste is as unbounded as the number of processes). Furthermore, a "garbage collection" in the sense of discarding vector components representing processes that in the meantime have been terminated is rendered virtually impossible (i.e. whenever any process with higher ID is still active and present in the clock vector in question).

For these reasons a somewhat more difficult, but much more flexible solution will be proposed here. The flexibility is achieved by investing some space and simplicity and making the clock vector a two-column matrix, variable in its number of rows, which provides a simple mapping of a process's ID to the associated (scalar) clock value. In this way, the size of the vector is no longer bounded by the (unbounded) number of processes but by two times the number of processes the maintaining process has (directly or indirectly) received clock values from, which can safely be assumed to be a significantly smaller number in most real-life applications (though, in fact, it is unbounded too). Furthermore, a "garbage collection" is possible by implementing a coordination protocol as motivated and described in the next section.

I will now describe in detail how these *dynamic vector clocks*[1] work and how they can be used for causal dependency/concurrency determination.

Each process p_i maintains its own logical clock C_i which is now a two-column matrix where $C_i(e_i^x)[k,2]$ holds, at the time of event e_i^x, the latest (scalar) clock value p_i knows from the process whose ID is saved in $C_i(e_i^x)[k,1]$. Initially, the clock consists of a single row with $C_i(e_i^0)[1,1] = i$ and $C_i(e_i^0)[1,2] = 0$ (supposing the clock index numbering starts with 1).

The update rules for dynamic vector clocks are specified as follows:

- If e_i^x is the receive event of message m, then $\forall l$ not exceeding the number of rows in $C_j(e_j^y)$: $C_i(e_i^x)[k,2] = max\{C_i(e_i^{x-1})[k,2], C_j(e_j^y)[l,2]\}$, where $C_j(e_j^y)$ is the clock matrix of the send event in process j that sent m to process i, and $C_j(e_j^y)[l,1] = C_i(e_i^x)[k,1]$.
- If there exists no k such that $C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1]$ for any l, then a new row is appended to $C_i(e_i^x)$ with $C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1]$ and $C_i(e_i^x)[k,2] = C_j(e_j^y)[l,2]$.
- If e_i^x is any event, including receive events, then, on receive events additionally to the above, the process increments its own scalar clock value: $C_i(e_i^x)[1,2] = C_i(e_i^{x-1})[1,2] + 1$ (with $C_i(e_i^x)[1,1] = i$ being the entry that represents the process itself as already postulated).

[1] I will not call them "matrix clocks", as this term has already been established for different concepts (for example in [6]).

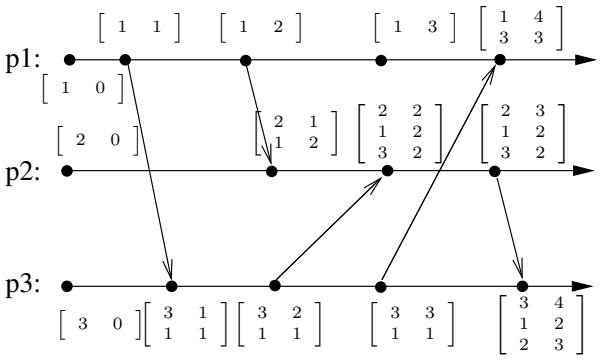

Figure 3: Dynamic vector clocks

An example illustrating the use of dynamic vector clocks is given in Figure 3. Note that the following important statement is always true:

$$\nexists k \neq l : C_i(e_i^x)[k,1] = C_i(e_i^x)[l,1].$$

This means that there is at most one entry representing a given process in any clock vector, which follows directly from the update rules: A new entry is only appended if there is no match already present. □

With these rules obeyed, the dynamic vector clocks retain all the benefits exhibited by standard vector clocks. However, before we examine the concurrency tests for dynamic vector clocks, we should redefine the $<$ relation, so it suits the strong clock condition:

Definition 3.6. The $<$ *relation for dynamic vector clocks* shall be defined as follows, with $|C|$ denoting the length of a clock in rows:

$$\begin{aligned} &C_i(e_i^x) < C_j(e_j^y) \text{ if and only if} \\ &\quad (C_i(e_i^x) \neq C_j(e_j^y)) \text{ and} \\ &\quad (\forall k \mid 0 < k \leq |C_i(e_i^x)| : (\exists l : 0 < l \leq |C_j(e_j^y)| \\ &\qquad \text{and } C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1] \\ &\qquad \text{and } C_i(e_i^x)[k,2] \leq C_j(e_j^y)[l,2])). \end{aligned}$$

For two dynamic vector clock values to be considered equal, the order of entries (rows) is certainly of no significance.

With the $<$ relation adapted, we now can state, in analogy to standard vector clocks, a simple non-symmetric causal dependency test:

$$\begin{aligned} &e_i^x \Rightarrow e_j^y \text{ if and only if } \exists k,l : \\ &\quad C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1] = i \text{ and} \\ &\quad C_i(e_i^x)[k,2] \leq C_j(e_j^y)[l,2]. \end{aligned}$$

Of course, analogous to standard vector clocks, a symmetric concurrency test can also be given:

$$e_i^x \parallel e_j^y \text{ if and only if}$$
$$C_j(e_j^y) \not\leq C_i(e_i^x) \text{ and}$$
$$C_i(e_i^x) \not\leq C_j(e_j^y).$$

The number of comparisons required for this test develops as of $O(|C_i(e_i^x)| \times |C_j(e_j^y)|)$. This time, there is no point in refining the test by reducing it to a comparison of two specific clock entries, because there are no fixed indices so finding these entries still involves searching the whole vector (and they might not even exist).

If this test is applied and yields $e_i^x \nparallel e_j^y$, it can be immediately determined, depending on which one of the two (highest level) conditional terms failed to be true, whether $e_i^x \Rightarrow e_j^y$ or $e_j^y \Rightarrow e_i^x$. This feature is the greatest benefit of vector clocks, and is retained as well by the dynamic extension presented herein.

Pruning Obsolete Clock Entries. Finally, an examination of the "garbage collection" possibilities mentioned at the beginning of this section. As opposed to the aforementioned somewhat simpler approach to render vector clocks suitable for dynamically changing systems, the extension proposed here is, in principle, suited for a "garbage collection" in the sense that the clocks can not only grow as needed, but also be pruned down by discarding information on processes that have already been terminated. The idea is, that whenever the maintaining process p_i is notified of the termination or destruction of process p_j, it can delete the entry representing p_j from its local clock (i.e. the entry with $C_i(e_i^x)[k,1] = j$) if present. Mostly, a notification of this kind would be propagated by an underlying system management. In this way, a lot of space, network load, and computation time (when comparing clocks values as described above) could be saved. However, for this to work some serious side effects have to be accounted for. For one thing, entries must be discarded equally on all clocks and timestamps that yet have to be evaluated. If two timestamps are compared, one of which has a given entry pruned while the other has not, concurrency/dependency checks might produce wrong results. Furthermore, entries representing a terminated process must not be pruned at all while there are still messages around which originate from that process and are possibly still needed for clock evaluation. Concurrency or dependency checks, as explained above, need the clock entries representing the two maintaining processes, respectively. Therefore these entries must not be discarded either from the timestamps on "orphaned" messages (which could easily exist due to network/routing delays) or from the timestamps or local clocks with which they might still be compared when finally delivered.

An example is given in Figure 4. The two messages arriving at p_3 would normally have to be viewed as causally ordered. However, if the first message is

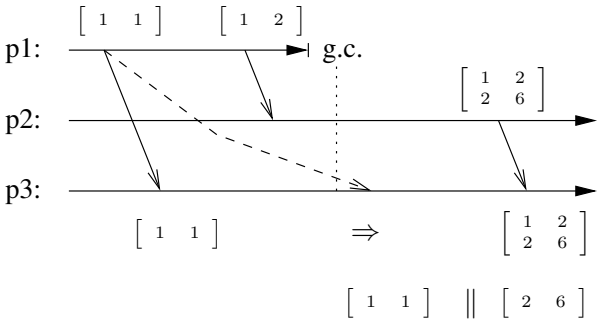

Figure 4: Faulty time stamp comparison after incautious garbage collection

delayed and arrives after a garbage collection which has deleted the first entry of the second message's time stamp, the two time stamps would possibly still have to be compared and the comparison would yield concurrency.

So heavy precaution must be taken by the system management to ensure that there are none of the terminated process's messages around any more whose timestamps could still have to be evaluated and would be corrupted by an early garbage collection.[1]

In the following, a garbage collection protocol is described which is intended to work under the conditions given in [2] for the implementation of causal delivery using vector clocks (see also 3.2). It is assumed that a monitor process p_m is notified by process p_j if p_j is terminating and that p_m is notified of all relevant events in the system, so this one process is able to order the incoming notifications properly according to their causality (i.e. p_m needs causal delivery, see section 3 and [2]). Due to the causal delivery p_m only reads the termination notification of p_j after all other event notifications of the same process have been accounted for. So, at this point, p_m can be sure to know all events of p_j up to the termination. Then the following steps are performed to assure that all timestamps are pruned at the same logical time. First p_m (or some other management module which is provided the necessary information by p_m) broadcasts a message m_1 to all remaining processes[2] which tells them to continue receiving messages but stop sending anything other than the receive notifications addressed to p_m. Now all processes have to confirm the receipt of m_1 to p_m and p_m waits for all confirmations to have arrived. Again, because of the causal delivery, which otherwise would have prevented p_m from receiving all confirmation messages, p_m now knows about all send events that have been carried out in the system so far and waits until the number of receive notifications equals the number of send events, which assures that there are no more messages in transit. p_m then broadcasts a message m_2 instructing all recipients to delete specific entries from their clocks and awaits

[1] Just having p_3 retroactively prune the p_1 entry on receiving the delayed message would seem convenient in this specific example, but is not, for instance, feasible in reverse case examples.

[2] The Processes are assumed to be known at least to some part of the system management at runtime.

all confirmations. Now that all clocks in the system are consistent again, p_m can broadcast a message m_3 telling the processes that it is safe to resume sending. Note that the only process for which causal delivery is assumed is p_m. In this protocol $5n$ additional messages have to be sent for the garbage collection, n denoting the number of processes, which makes a linear complexity.

Proof of correctness: First, the protocol has to assure that p_m knows of all messages sent by the terminated process p_j. Assume that p_j has sent a message p_m is still not aware of. Because p_m must have been notified about the sending of the message in question it must be the case that p_m still has not read the notification message. But in this case, due to causal delivery, it also can not have read the termination notification of p_j which is causally dependent on all of p_j's send events. So the assumption is contradictory.

Now, all send events have to be prohibited. In analogy to the above, p_m knows of all send events of all active processes p_i up to the moment it receives p_i's confirmation message. This assures p_m is able to count incoming receive notifications (which still are allowed to be sent) and compare them with the number of sent messages, which has to be constant due to the sending prohibition. Since messages are delivered reliably and with finite delay (see section 2), the number receive notifications arrived at p_m finally must reach the number of the messages sent. At this point, p_m can conclude there is no more message in transit. Otherwise, p_m could not have received a receive notification for the message. For the numbers of sent and received messages to be equal p_m also would have to lack the notification of the sending of the message. This, in turn, would only have been possible if p_m did not receive one of the messages confirming its initiation of the sending prohibition (because of causal delivery). But these are explicitly awaited.

So, no messages are in transit when the entries referring to the terminated process are globally deleted. Also, no more messages can be sent for which these entries could be of importance, since the process the entries represent is already terminated. □

4 Conclusion

In the paper at hand logical clocks as the most commonly used devices for logically ordering events in distributed systems have been discussed in detail and algorithms for their appliance to the most important event ordering problems have been given. In particular, an extension of vector clocks has been proposed and integrated that is meant to enable the use of vector clocks, while retaining their well established concept, in environments featuring a dynamically changing number of processes. Efficiency issues introduced by the concept of these *dynamic vector clocks* have been examined, as has a solution idea based on discarding obsolete clock entries. An appropriate garbage collection protocol has been proposed. The examination shows that the tracking of logical time in dynamically growing/shrinking applications can be done using the dynamic vector clocks introduced in this paper with some additional expenses as compared to standard vector clocks. The only one of these expenses which must be considered critical is the fact that, in the worst case, pruning the dynamic vector clocks involves halting the entire computation. Therefore, a trade-off between the size of time stamps and the frequency of performing the garbage collection has to be carefully considered. The protocol allows for the clock entries of several terminated processes to be pruned in one garbage collection. Thus it is possible, for example, to postpone each garbage collection until n processes have terminated. Increasing n would then result in fewer garbage collections at the expense of tolerating time stamps to contain up to $n-1$ obsolete entries (consisting of two integers each).

Mind that the results presented are to be applied only to the "general" case. A minimal set of assumptions has been made regarding the architecture and behaviour of the processes and the underlying system. Further work will have to show what improvements can be made by considering a more specific system behaviour.

Acknowledgments

I would like to thank Jörg Preißinger for the helpful discussions during my work on the contents presented in this paper.

References

[1] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. In *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978.

[2] Özalp Babaoğlu, Keith Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison Wesley, 2nd edition, 1993.

[3] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard et al., editor, *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226, Elsevier Science Publishers B.V., North-Holland, 1989.

[4] Colin Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 55–66, February 1988.

[5] Colin Fidge. Logical Time in Distributed Computer Systems. In *Computer*, 24(8), pages 28–33, August 1991.

[6] Igor A. Zhuklinets, D. A. Khotimsky. Logical Time in Distributed Software Systems. In *Programming and Computing Software*, Volume 28, Issue 3, pages 174–184, Plenum Press, May 2002.

[7] Reinhard Schwarz, Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. In *Distributed Computing*, 7(3): pages 149–174, 1994.