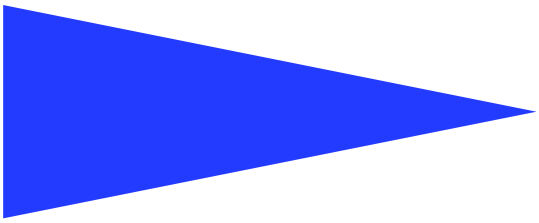
I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

P U B L I C A T I O N
I N T E R N E
N° 900



LOGICAL TIME: A WAY TO CAPTURE CAUSALITY IN DISTRIBUTED SYSTEMS

M. RAYNAL, M. SINGHAL

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

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

Logical Time: A Way to Capture Causality in Distributed Systems

M. Raynal*, M. Singhal**

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

Publication interne n ° 900 — Janvier 1995 — 22 pages

Abstract: The concept of causality between events is fundamental to the design and analysis of parallel and distributed computing and operating systems. Usually causality is tracked using physical time, but in distributed systems setting, there is no built-in physical time and it is only possible to realize an approximation of it. As asynchronous distributed computations make progress in spurts, it turns out that the logical time, which advances in jumps, is sufficient to capture the fundamental monotonicity property associated with causality in distributed systems. This paper reviews three ways to define logical time (e.g., scalar time, vector time, and matrix time) that have been proposed to capture causality between events of a distributed computation.

Key-words: Distributed systems, causality, logical time, happens before, scalar time, vector time, matrix time.

(Résumé : tsvp)

*IRISA, Campus de Beaulieu, 35042 Rennes-Cédex, raynal@irisa.fr.
**Dept. of Computer, Information Science, Columbus, OH 43210, singhal@cis.ohio-state.edu.

Centre National de la Recherche Scientifique
(URA 227) Université de Rennes 1 – Insa de Rennes
INRIA
Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Le temps logique en réparti ou comment capturer la causalité

Résumé : Ce rapport examine différents mécanismes d'horlogerie logique qui ont été proposés pour capturer la relation de causalité entre événements d'un système réparti.

Mots-clé : causalité, précédence, sytème réparti, temps logique, temps linéaire, temps vectoriel, temps matriciel.

1 Introduction

A distributed computation consists of a set of processes that cooperate and compete to achieve a common goal. These processes do not share a common global memory and communicate solely by passing messages over a communication network. The communication delay is finite but unpredictable. The actions of a process are modeled as three types of events, namely, internal events, message send events, and message receive events. An internal event only affects the process at which it occurs, and the events at a process are linearly ordered by their order of occurrence. Moreover, send and receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process. It follows that the execution of a distributed application results in a set of distributed events produced by the processes. The causal precedence relation induces a partial order on the events of a distributed computation.

Causality (or the causal precedence relation) among events in a distributed system is a powerful concept in reasoning, analyzing, and drawing inferences about a computation. The knowledge of the causal precedence relation among processes helps solve a variety of problems in distributed systems. Among them we find:

- Distributed algorithms design: The knowledge of the causal precedence relation among events helps ensure liveness and fairness in mutual exclusion algorithms, helps maintain consistency in replicated databases, and helps design correct deadlock detection algorithms to avoid phantom and undetected deadlocks.
- Tracking of dependent events: In distributed debugging, the knowledge of the causal dependency among events helps construct a consistent state for resuming reexecution; in failure recovery, it helps build a checkpoint; in replicated databases, it aids in the detection of file inconsistencies in case of a network partitioning.
- Knowledge about the progress: The knowledge of the causal dependency among events helps a process measure the progress of other processes in the distributed computation. This is useful in discarding obsolete information, garbage collection, and termination detection.
- Concurrency measure: The knowledge of how many events are causally dependent is useful in measuring the amount of concurrency in a computation. All events that are not causally related can be executed concurrently. Thus, an analysis of the causality in a computation gives an idea of the concurrency in the program.

The concept of causality is widely used by human beings, often unconsciously, in planning, scheduling, and execution of a chore or an enterprise, in determining infeasibility of a plan or the innocence of an accused. In day-today life, the global time to deduce causality relation is obtained from loosely synchronized clocks (i.e., wrist watches, wall clocks). However, in distributed computing systems, the rate of occurrence of events is several magnitudes higher and the event execution time is several magnitudes smaller; consequently, if the physical clocks are not precisely synchronized, the causality relation between events may not be accurately captured. However, in a distributed computation, progress is made in spurts and the interaction between processes occurs in spurts; consequently, it turns out that in a distributed computation, the causality relation between events produced by a program execution, and its fundamental monotonicity property, can be accurately captured by logical clocks.

In a system of logical clocks, every process has a logical clock that is advanced using a set of rules. Every event is assigned a timestamp and the causality relation between events can be generally inferred from their timestamps. The timestamps assigned to events obey the fundamental monotonicity property; that is, if an event a causally affects an event b, then the timestamp of a is smaller than the timestamp of b.

This paper first presents a general framework of a system of logical clocks in distributed systems and then discusses three ways to implement logical time in a distributed system. In the first method, the Lamport's scalar clocks, the time is represented by non-negative integers; in the second method, the time is represented by a vector of non-negative integers; in the third method, the time is represented as a matrix of non-negative integers. The rest of the paper is organized as follows: The next section presents a model of the execution of a distributed computation. Section 3 presents a general framework of logical clocks as a way to capture causality in a distributed computation. Sections 4 through 6 discuss three popular systems of logical clocks, namely, scalar, vector, and matrix clocks. Section 7 discusses efficient implementations of the systems of logical clocks. Finally Section 8 concludes the paper.

2 A Model of Distributed Executions

2.1 General Context

A distributed program is composed of a set of n asynchronous processes p_1, p_2, ..., p_i, ..., p_n that communicate by message passing over a communication network. The

processes do not share a global memory and communicate solely by passing messages. The communication delay is finite and unpredictable. Also, these processes do not share a global clock that is instantaneously accessible to these processes. Process execution and a message transfer are asynchronous – a process may execute an event spontaneously and a process sending a message does not wait for the delivery of the message to be complete.

2.2 Distributed Executions

The execution of process p_i produces a sequence of events e_i^0, e_i^1, ..., e_i^x, e_i^{x+1}, ... and is denoted by $\mathcal{H}_i$ where

$$\mathcal{H}_i = (h_i, \rightarrow_i)$$

h_i is the set of events produced by p_i and binary relation $\rightarrow_i$ defines a total order on these events. Relation $\rightarrow_i$ expresses causal dependencies among the events of p_i.

A relation $\rightarrow_{msg}$ is defined as follows. For every message m that is exchanged between two processes, we have

$$send(m) \rightarrow_{msg} receive(m).$$

Relation $\rightarrow_{msg}$ defines causal dependencies between the pairs of corresponding send and receive events.

A distributed execution of a set of processes is a partial order $\mathcal{H}$=(H, $\rightarrow$), where

$$H = \cup_i h_i \text{ and } \rightarrow = (\cup_i \rightarrow_i \cup \rightarrow_{msg})^+.$$

Relation $\rightarrow$ expresses causal dependencies among the events in the distributed execution of a set of processes. If $e_1 \rightarrow e_2$, then event e_2 is directly or transitively dependent on event e_1. If $e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$, then events e_1 and e_2 are said to be concurrent and are denoted as $e_1 \parallel e_2$. Clearly, for any two events e_1 and e_2 in a distributed execution, $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$, or $e_1 \parallel e_2$.

Figure 1 shows the time diagram of a distributed execution involving three processes. A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer. In this execution, $a \rightarrow b$, $b \rightarrow d$, and $b \parallel c$.

2.3 Distributed Executions at an Observation Level

Generally, at a level or for an application, only few events are relevant. For example, in a checkpointing protocol, only local checkpoint events are relevant. Let R denote

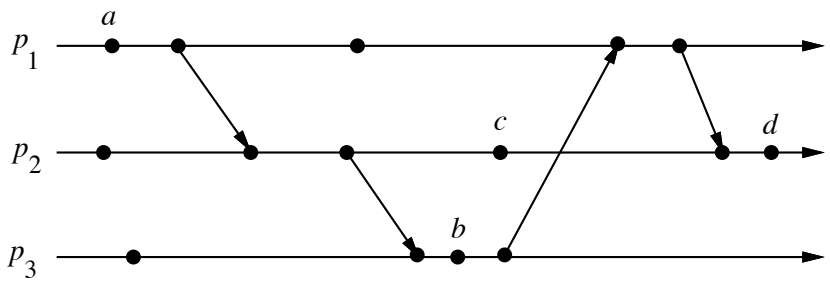


Figure 1: The time diagram of a distributed execution.

the set of relevant events. Let $\rightarrow_R$ be the restriction of $\rightarrow$ to the events in R; that is,

$$\forall\ e_1, e_2 \in R\text{: } e_1 \rightarrow_R e_2 \iff e_1 \rightarrow e_2.$$

An observation level defines a projection of the events in the distributed computation. The distributed computation defined by the observation level R is denoted as,

$$\mathcal{R}=(R, \rightarrow_R).$$

For example, if in Figure 1, only events a, b, c, and d are relevant to an observation level (i.e., $R=\{a, b, c, d\}$), then relation $\rightarrow_R$ is defined as follows: $\rightarrow_R= \{(a, b), (a, c), (a, d), (b, d), (c, d)\}$.

3 Logical Clocks: A Mechanism to Capture Causality

3.1 Definition

A system of logical clocks consists of a time domain T and a logical clock C. Elements of T form a partially ordered set over a relation $<$. This relation is usually called *happened before* or *causal precedence*; intuitively this relation is the analogous of the *earlier than* relation provided by physical time. The logical clock C is a function that maps an event e in a distributed system to an element in the time domain T, denoted as C(e) and called the timestamp of e, and is defined as follows:

$$C : H \mapsto T$$

such that the following property is satisfied:

$$e_1 \rightarrow e_2 \Longrightarrow \mathrm{C}(e_1) < \mathrm{C}(e_2).$$

This monotonicity property is called the *clock consistency condition.* When T and C satisfy the following condition,

$$e_1 \rightarrow e_2 \Longleftrightarrow \mathrm{C}(e_1) < \mathrm{C}(e_2)$$

the system of clocks is said to be *strongly consistent.*

3.2 Implementing Logical Clocks

Implementation of logical clocks requires addressing two issues: data structures local to every process to represent logical time and a protocol (set of rules) to update the data structures to ensure the consistency condition.

Each process p_i maintains data structures that allow it the following two capabilities:

- A *local logical clock*, denoted by lc_i, that helps process p_i measure its own progress.
- A *logical global clock*, denoted by gc_i, that is a good representation of process p_i's local view of the logical global time. It allows this process to assign consistent timestamps to its local events. Typically, lc_i is a part of gc_i.

The protocol ensures that a process's logical clock and thus its view of the global time is managed consistently. The protocol consists of the following two rules:

- *R1*: This rule governs how the local logical clock is updated by a process (to capture its progress) when it executes an event (send, receive, or internal).
- *R2*: This rule governs how a process updates its global logical clock to update its view of the global time and global progress. It dictates what information about the logical time is piggybacked in a message and how this information is used by the receiving process to update its view of the global time.

Systems of logical clocks differ in their representation of logical time and also in the protocol to update the logical clocks. However, all logical clock systems implement the rules *R1* and *R2* and consequently ensure the fundamental monotonicity property associated with causality. Moreover each particular logical clock system provides its users with some additional properties.

4 Scalar Time

4.1 Definition

The scalar time representation was proposed by Lamport in 1978 [7] as an attempt to totally order events in a distributed system. Time domain in this representation is the set of non-negative integers. The logical local clock of a process p_i and its local view of the global time are squashed into one integer variable C_i.

Rules *R1* and *R2* to update the clocks are as follows:

- *R1*: Before executing an event (send, receive, or internal), process p_i executes the following:
$$C_i := C_i + d \qquad (d > 0)$$
(each time *R1* is executed d can have a different value).
- *R2*: Each message piggybacks the clock value of its sender at sending time. When a process p_i receives a message with timestamp C_{msg}, it executes the following actions:
  - $C_i := max(C_i, C_{msg})$
  - Execute *R1*.
  - Deliver the message.

Figure 2 shows evolution of scalar time with d=1 for the computation in Figure 1.

4.2 Basic Property

Clearly, scalar clocks satisfy the monotonicity and hence the consistency property. In addition, scalar clocks can be used to totally order the events in a distributed system as follows [7]: The timestamp of an event is denoted by a tuple (t, i) where t is its time of occurrence and i is the identity of the process where it occurred. The total order relation $\prec$ on two events x and y with timestamps *(h,i)* and *(k,j)*, respectively, is defined as follows:

$$x \prec y \iff (h < k \textbf{ or } (h = k \textbf{ and } i < j))$$

The total order is consistent with the causality relation "$\rightarrow$". The total order is generally used to ensure liveness properties in distributed algorithms (requests are timestamped and served according to the total order on these timestamps) [7].

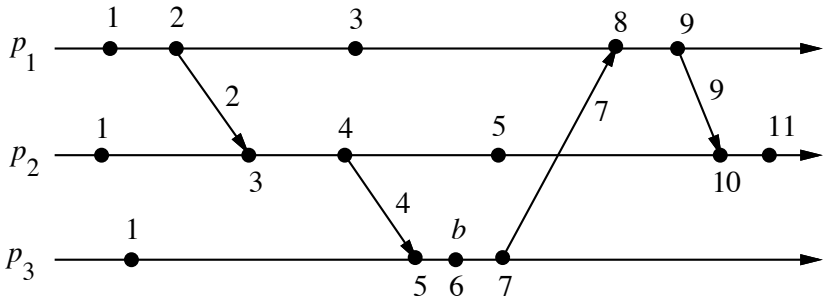


Figure 2: Evolution of scalar time.

If the increment value d is always 1, the scalar time has the following interesting property: if event e has a timestamp h, then *h-1* represents the minimum logical duration, counted in units of events, required before producing the event e [2]; we call it the height of the event e, in short *height(e)*. In other words, *h-1* events have been produced sequentially before the event e regardless of the processes that produced these events. For example, in Figure 2, five events precede event b on the longest causal path ending at b.

However, system of scalar clocks is not strongly consistent; that is, for two events e_1 and e_2, $C(e_1) < C(e_2) \not\Longrightarrow e_1 \rightarrow e_2$. Reason for this is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss of information.

5 Vector Time

5.1 Definition

The systems of vector clocks was developed independently by Fidge [2,3], Mattern [8], and Schmuck [10]. (A brief historical perspective of vector clocks is given in the Sidebar 2.) In the system of vector clocks, the time domain is represented by a set of n-dimensional non-negative integer vectors. Each process p_i maintains a vector $vt_i[1..n]$ where $vt_i[i]$ is the local logical clock of p_i and describes the logical time progress at process p_i. $vt_i[j]$ represents process p_i's latest knowledge of process p_j local time. If $vt_i[j]=x$, then process p_i knows that local time at process p_j has

progressed till x. The entire vector vt_i constitutes p_i's view of the logical global time and is used to timestamp events.

Process p_i uses the following two rules *R1* and *R2* to update its clock:

- *R1*: Before executing an event, it updates its local logical time as follows:

$$vt_i[i] := vt_i[i] + d \qquad (d > 0)$$

- *R2*: Each message m is piggybacked with the vector clock vt of the sender process at sending time. On the receipt of such a message *(m,vt)*, process p_i executex the following sequence of actions:
  - Update its logical global time as follows:

$$1 \leq k \leq n \ : \ vt_i[k] := max(vt_i[k], vt[k])$$

  - Execute *R1*.
  - Deliver the message m.

The timestamp associated with an event is the value of the vector clock of its process when the event is executed. Figure 3 shows an example of vector clocks progress with the increment value *d=1*.

5.2 Basic Property

Isomorphism

The following three relations are defined to compare two vector timestamps, vh and vk:

$$\begin{aligned} vh \leq vk &\iff \forall\ x : vh[x] \leq vk[x] \\ vh < vk &\iff vh \leq vk \textbf{ and } \exists\ x : vh[x] < vk[x] \\ vh \parallel vk &\iff \textbf{not } (vh < vk) \textbf{ and not } (vk < vh) \end{aligned}$$

Recall that relation "$\rightarrow$" induces a partial order on the set of events that are produced by a distributed execution. If events in a distributed system are timestamped using a system of vector clocks, we have the following property.

If two events x and y have timestamps vh and vk, respectively, then:

$$\begin{aligned} x \rightarrow y &\iff vh < vk \\ x \parallel y &\iff vh \parallel vk. \end{aligned}$$

Figure 3: Evolution of vector time.

Thus, there is an isomorphism between the set of partially ordered events produced by a distributed computation and their timestamps. This is a very powerful, useful, and interesting property of vector clocks. If processes of occurrence of events are known, the test to compare two timestamps can be simplified as follows:

If events x and y respectively occurred at processes p_i and p_j and are assigned timestamps *(vh,i)* and *(vk,j)*, respectively, then

$$\begin{aligned} x \rightarrow y &\iff vh[i] < vk[i] \\ x \parallel y &\iff vh[i] > vk[i] \textbf{ and } vh[j] < vk[j] \end{aligned}$$

Strong Consistency

The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related. However, the dimension of vector clocks cannot be less than n for this property [1].

Event Counting

If d is always 1 in the rule *R1*, then the i^{th} component of vector clock at process p_i, $vt_i[i]$, denotes the number of events that have occurred at p_i until that instant. So, if an event e has timestamp vh, $vh[j]$ denotes the number of events executed by

process p_j that causally precede e. Clearly, $\sum vh[j] - 1$ represents the total number of events that causally precede e in the distributed computation.

Applications

Since vector time tracks causal dependencies exactly, it finds a wide variety of applications. For example, they are used in distributed debugging, in the implementations of causal ordering communication and causal distributed shared memory, in establishing global breakpoints, and in determining the consistency of checkpoints in optimistic recovery.

6 Matrix Time

6.1 Definition

In a system of matrix clocks, the time is represented by a set of $n \times n$ matrices of non-negative integers. A process p_i maintains a matrix $mt_i[1..n, 1..n]$ where

- $mt_i[i, i]$ denotes the local logical clock of p_i and tracks the progress of the computation at process p_i.
- $mt_i[k, l]$ represents the knowledge that process p_i has about the knowledge of p_k about the logical local clock of p_l. The entire matrix mt_i denotes p_i's local view of the logical global time.

Note that row $mt_i[i, .]$ is nothing but the vector clock $vt_i[.]$ and exhibits all properties of vector clocks.

Process p_i uses the following rules *R1* and *R2* to update its clock:

- *R1* : Before executing an event, it updates its local logical time as follows:

$$mt_i[i, i] := mt_i[i, i] + d \qquad (d > 0)$$

- *R2*: Each message m is piggybacked with matrix time mt. When p_i receives such a message *(m,mt)* from a process p_j, p_i executes the following sequence of actions:
  - Update its logical global time as follows:

$$1 \leq k \leq n : mt_i[i, k] := max(mt_i[i, k], mt[j, k])$$
$$1 \leq k, l \leq n : mt_i[k, l] := max(mt_i[k, l], mt[k, l])$$

- Execute *R1*.
- Deliver message m.

A system of matrix clocks was first informally proposed by Michael and Fischer in 1982 [4] and has been used by Wuu and Bernstein [12] and by Lynch and Sarin [9] to discard obsolete information in replicated databases.

6.2 Basic Property

Clearly vector $mt_i[i, .]$ contains all the properties of vector clocks. In addition, matrix clocks have the following property:

$$\min_k(mt_i[k, l]) \geq t \Rightarrow \quad \text{process } p_i \text{ knows that every other process } p_k \text{ knows that } p_l\text{'s local time has progressed till } t$$

If this is true, it is clear that process p_i knows that all other processes know that p_l will never send information with a local time $\leq t$. In many applications, this implies that processes will no longer require from p_l certain information and can use this fact to discard obsolete information.

If d is always 1 in the rule *R1*, then $mt_i[k, l]$, denotes the number of events occurred at p_l and known by p_k as far as p_i's knowledge is concerned.

7 Efficient Implementations

If the number of processes in a distributed computation is large, then vector and matrix clocks will require piggybacking of huge information in messages for the purpose of disseminating time progress and updating clocks. In this section, we discuss efficient ways to maintain vector clocks; similar techniques can be used to efficiently implement matrix clocks.

It has been shown [1] that if vector clocks have to satisfy the strong consistency property, then in general vector timestamps must be at least of size n. Therefore, in general the size of vector timestamp is the number of processes involved in a distributed computation; however, several optimizations are possible and next, we discuss techniques to implement vector timestamps efficiently.

7.1 Singhal-Kshemkalyani's Differential Technique

Singhal-Kshemkalyani's technique [11] is based on the observation that between successive events at a process, only few entries of the vector clock at that process are

likely to change. This is more true when the number of processes is large because only few of them will interact frequently by passing messages. In Singhal-Kshemkalyani's differential technique, when a process p_i sends a message to a process p_j, it piggybacks only those entries of its vector clock that differ since the last message send to p_j. Therefore, this technique cuts down the communication bandwidth and buffer (to store messages) requirements. However, a process needs to maintain two additional vectors to store the information regarding the clock values when the last interaction was done with other processes.

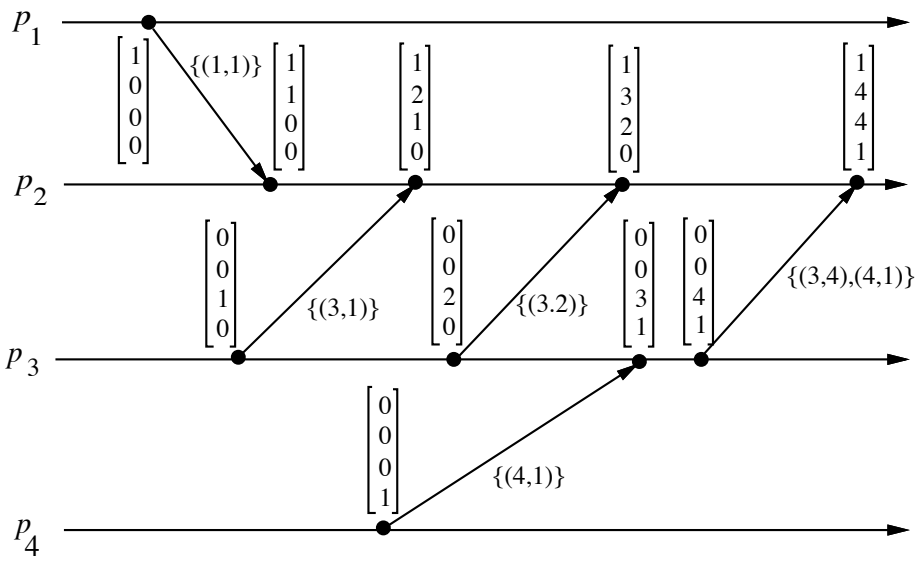


Figure 4: Vector clocks progress in Singhal-Kshemkalyani technique

Figure 4 illustrates the Singhal-Kshemkalyani technique. If entries i_1, i_2, ..., i_{n1} of the vector clock at p_i have changed (to v_1, v_2, ..., v_{n1}, respectively) since the last message send to p_j, then process p_i piggybacks a compressed timestamp of the from $\{(i_1, v_1), (i_2, v_2), ..., (i_{n1}, v_{n1}))\}$ to the next message to p_j. When p_j receives this message, it updates its clock as follows: $vt_i[k]:= max(vt_i[k], v_k)$ for k=1, 2, ..., n1. This technique can substantially reduce the cost of maintaining vector clocks in large systems if process interaction exhibits temporal or spatial localities. However, it requires that communication channels be FIFO.

7.2 Fowler-Zwaenepoel's Direct-Dependency Technique

Fowler-Zwaenepoel's direct-dependency technique [5] does not maintain vector clocks on-the-fly. Instead, in this technique a process only maintains information regarding direct dependencies on other processes. A vector time for an event, that represents transitive dependencies on other processes, is constructed off-line from a recursive search of the direct dependency information at processes. A process p_i maintains a dependency vector D_i that is initially $D_i[j]=0$ for $j=1..n$ and is updated as follows:

- Whenever an event occurs at p_i: $D_i[i]:=D_i[i]+1$.
- When a process p_j sends a message m to p_i, it piggybacks the updated value of $D_j[j]$ in the message.
- When p_i receives a message from p_j with piggybacked value d, p_i updates its dependency vector as follows: $D_i[j]:= max\{D_i[j], d\}$.

Thus, the dependency vector at a process only reflects direct dependencies. At any instant, $D_i[j]$ denotes the sequence number of the latest event on process p_j that *directly* affects the current state. Note that this event may precede the latest event at p_j that *causally* affects the current state.

Figure 5 illustrates the Fowler-Zwaenepoel technique. This technique results in considerable saving in the cost; only one scalar is piggybacked on every message. However, the dependency vector does not represent transitive dependencies (i.e., a vector timestamps). The transitive dependency (or the vector timestamp) of an event is obtained by recursively tracing the direct-dependency vectors of processes. Clearly, this will have overhead and will involve latencies. Therefore, this technique is not suitable for applications that require on-the-fly computation of vector timestamps. Nonetheless, this technique is ideal for applications where computation of causal dependencies is performed off-line (e.g., causal breakpoint, asynchronous checkpointing recovery).

7.3 Jard-Jourdan's Adaptive Technique

In the Fowler-Zwaenepoel's direct-dependency technique, a process must observe an event (i.e., update and record its dependency vector) after receiving a message but before sending out any message. Otherwise, during the reconstruction of a vector timestamp from the direct-dependency vectors, all the causal dependencies will not be captured. If events occur very frequently, this technique will require recording the history of a large number of events. In the Jard-Jourdan's technique [6], events can

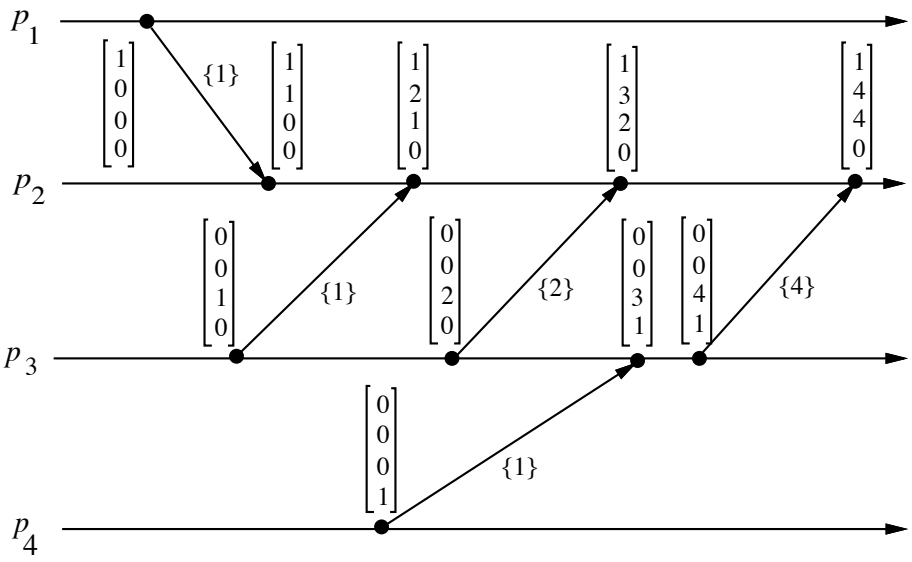


Figure 5: Vector clocks progress in Fowler-Zwaenepoel technique

be adaptively observed while maintaining the capability of retrieving all the causal dependencies of an observed event.

Jard-Jourdan define a *pseudo-direct* relation $\ll$ on the events of a distributed computation as follows: if events e_i and e_j happen at processes p_i and p_j, respectively, then $e_j \ll e_i$ iff there exists a path of message transfers that starts after e_j on p_j and ends before e_i on p_i such that there is no observed event on the path.

The partial vector clock p_vt_i at process p_i is a list of tuples of the form $(j,\ v)$ indicating that the current state of p_i is pseudo-dependent on the event on process p_j whose sequence number is v. Initially, at a process p_i: p_vt_i=$\{(i,\ 0)\}$.

- Whenever an event is observed at process p_i, the following actions are executed (Let p_vt_i =$\{(i1, v1)$,...,(i, v), ...$\}$, denote the current partial vector clock at p_i and variable e_vt_i holds the timestamp of the observed event):
  - $e\_vt_i = \{(i1, v1)$,...,(i, v), ...$\}$.
  - $p\_vt_i := \{(i, v+1)\}$.
- When process p_j sends a message to p_i, it piggybacks the current value of p_vt_j in the message.

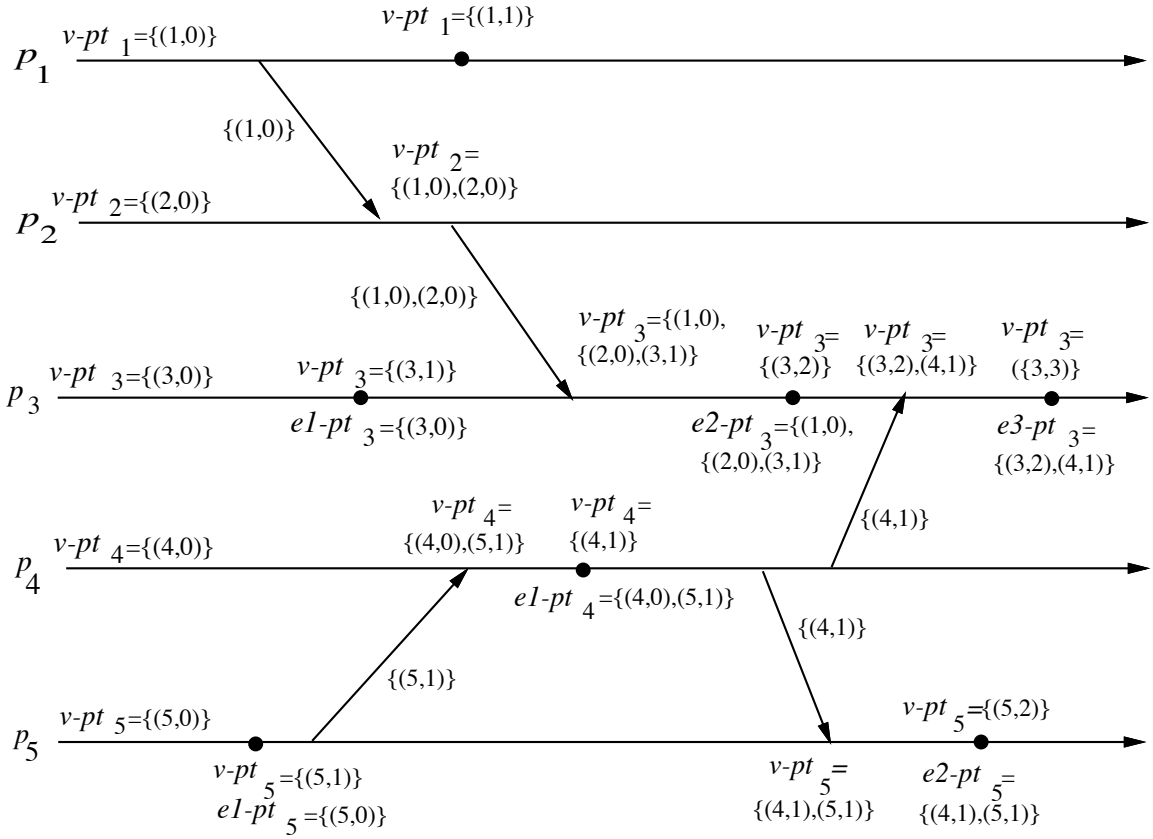


Figure 6: Vector clocks progress in Jard-Jourdan technique

- When p_i receives a message piggybacked with timestamp p_vt, p_i updates p_vt_i such that it is the union of the following: (Let p_vt={(i_{m1}, v_{m1}),...,(i_{mk}, v_{mk})} and p_vt_i={(i_1, v_1),,...,(i_l, v_l)}.)
  - all (i_{mx}, v_{mx}) such that $(i_{mx}, .)$ does not appear in v_pt_i,
  - all (i_x, v_x) such that $(i_x, .)$ does not appear in v_pt, and
  - all $(i_x, max(v_x, v_{mx}))$ for all $(v_x, .)$ that appear in v_pt and v_pt_i.

Figure 6 illustrates the Jard-Jourdan technique to maintain vector clocks. eX_pt_n denotes the timestamp of the Xth observed event at process p_n. For example, the third event observed at p_i is timestamped $e3\_pt_3$={(3,2,(4,1)}; this timestamp means that the pseudo-direct predecessors of this event are located at processes p_3 and p_4

and are respectivley the second event observed at p_3 and the first observed at p_4. So, considering the timestamp of an event, the set of the observed events that are its predecessors can be easily computed.

8 Concluding Remarks

The concept of causality between events is fundamental to the design and analysis of distributed programs. The notion of time is basic to capture causality between events; however, there is no built-in physical time in distributed systems and it is only possible to realize an approximation of it. But a distributed computation makes progress in spurts and consequently logical time, which advances in jumps, is sufficient to capture the monotonicity property induced by causality in distributed systems.

This paper presented a general framework of logical clocks in distributed systems and discussed three systems of logical clocks, namely, scalar, vector, and matrix, that have been proposed to capture causality between events of a distributed computation. We discussed properties and applications of these systems of clocks. The power of a system of clocks increases in this order, but so do the complexity and the overheads. We discussed efficient implementations of these systems of clocks.

References

[1] Charron-Bost, B. *Concerning the size of logical clocks in distributed systems.* Inf. Proc. Letters, vol.39, (1991), pp. 11-16.

[2] Fidge, L.J. *Timestamp in message passing systems that preserves partial ordering.* Proc. 11th Australian Comp. Conf., (Feb. 1988), pp. 56-66.

[3] Fidge, C. *Logical time in distributed computing systems.* IEEE Computer, (August 1991), pp. 28-33.

[4] Fischer, M.J., Michael, A. *Sacrifying serializability to attain hight availability of data in an unreliable network.* Proc. of ACM Symposium on Principles of Database Systems, (1982), pp. 70-75.

[5] Fowler J., Zwaenepoel W. *Causal distributed breakpoints.* Proc. of 10th Int'l. Conf. on Distributed Computing Systems, (1990), pp. 134-141.

[6] Jard C., Jourdan G-C. *Dependency tracking and filtering in distributed computations.* in Brief announcements of the ACM symposium on PODC, (1994). (A full presentation appeared as IRISA Tech. Report No. 851, 1994).

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

[8] Mattern, F. *Virtual time and global states of distributed systems.* Proc. "Parallel and distributed algorithms" Conf., (Cosnard, Quinton, Raynal, Robert Eds), North-Holland, (1988), pp. 215-226.

[9] Sarin, S.K., Lynch, L. *Discarding obsolete information in a replicated data base system.* IEEE Trans. on Soft. Eng., vol.SE 13,1, (Jan. 1987), pp. 39-46.

[10] Schmuck, F. *The use of efficient broadcast in asynchronous distributed systems.* Ph. D. Thesis, Cornell University, TR88-928, (1988), 124 pages.

[11] Singhal, M., Kshemkalyani, A. *An Efficient Implementation of Vector Clocks.* Information Processing Letters, 43, August 1992, pp. 47-52.

[12] Wuu, G.T.J., Bernstein, A.J. *Efficient solutions to the replicated log and dictionary problems.* Proc. 3rd ACM Symposium on PODC, (1984), pp. 233-242

Annex 1: Virtual Time

The *synchronizer* concept of Awerbuch [1] allows a synchronous distributed algorithm or program to run on an asynchronous distributed system. A synchronous distributed program executes in a lock-step manner and its progress relies on a global time assumption. A global time preexists in the semantics of synchronous distributed programs and it participates in the execution of such programs. A synchronizer is an interpreter for synchronous distributed programs and simulates a global time for such programs in an asynchronous environment [6].

In distributed, discrete-event simulations [5,8], a global time (the so-called the simulation time) exists and the semantics of a simulation program relies on such a time and its progress ensures that the simulation program has the liveness property. In the execution of a distributed simulation, it must be ensured that the virtual time progresses (liveness) in such a way that causality relations of the simulation program are never violated (safety).

The global time built by a synchronizer or by a distributed simulation runtime environment drives the underlying program and should not be confused with the logical time presented previously. The time provided by a synchronizer or a distributed simulation runtime support does belong to the underlying program semantics and is nothing but the virtual [4] counterpart of the physical time offered by the environment and used in real-time applications [2].

On the other hand, in the previous representations of logical time (e.g., linear, vector, or matrix), the aim is to order events according to their causal precedence in order to ensure some properties such as liveness, consistency, fairness, etc. Such a logical time is only a means among others to ensure some properties. For example Lamport's logical clocks are used in Ricart-Agrawala's mutual exclusion algorithm [7] to ensure liveness; this time does not belong to the mutual exclusion semantics or the program invoking mutual exclusion. In fact, other means can be used to ensure properties such as liveness; for example, Chandy and Misra's mutual exclusion algorithm [3] uses a dynamic, directed, acyclic graph to ensure liveness.

References

[1] Awerbuch, B. *Complexity of network synchronization.* Journal of the ACM, vol.32,4, (1985), pp. 804-823.

[2] Berry, G. *Real time programming : special purpose or general purpose languages.* IFIP Congress, Invited talk, San Francisco, (1989).

[3] Chandy, K.M., Misra, J. *The drinking philosophers problem.* ACM Toplas, vol.6,4, (1984), pp. 632-646.

[4] Jefferson, D. *Virtual time.* ACM Toplas, vol.7,3, (1985), pp. 404-425.

[5] Misra, J. *Distributed discrete event simulation.* ACM Computing Surveys, vol.18,1, (1986), pp. 39-65.

[6] Raynal, M., Helary, J.M. *Synchronization and control of distributed systems and programs.* Wiley & sons, (1990), 124 p.

[7] Ricart, G., Agrawala, A.K. *An optimal algorithm for mutual exclusion in computer networks.* Comm. ACM, vol.24,1, (Jan. 1981), pp. 9-17.

[8] Righter, R., Walrand, J.C. *Distributed simulation of discrete event systems.* Proc. of the IEEE, (Jan. 1988), pp. 99-113.

Annex 2: Vector Clocks - A Brief Historical Perspective

Although the theory associated with vector clocks was first developed in 1988 independently by Fidge, Mattern, and Schmuck, vector clocks were informally introduced and used by several researchers earlier. Parker *et al.* [2] used a rudimentary vector clocks system to detect inconsistencies of replicated files due to network partitioning. Liskov and Ladin [1] proposed a vector clock system to define highly available distributed services. Similar system of clocks was used by Strom and Yemini [4] to keep track of the causal dependencies between events in their optimistic recovery algorithm and by Raynal to prevent drift between logical clocks [3].

References

[1] Liskov, B., Ladin, R. *Highly available distributed services and fault-tolerant distributed garbage collection.* Proc. 5th ACM Symposium on PODC, (1986), pp. 29-39.

[2] Parker, D.S. *et al. Detection of mutual inconsistency in distributed systems.* IEEE Trans. on Soft. Eng., vol.SE 9,3, (May 1983), pp. 240-246.

[3] Raynal, M. *A distributed algorithm to prevent mutual drift between n logical clocks.* Inf. Processing Letters, vol.24, (1987), pp. 199-202.

[4] Strom, R.E., Yemini, S. *Optimistic recovery in distributed systems.* ACM TOCS, vol.3,3, (August 1985), pp. 204-226.