LOGICAL TIME:

Capturing Causality in Distributed Systems

Michel Raynal
University of Rennes

Mukesh Singhal
Ohio State University

Causality—determining which event happens before what others—is vital in distributed computations. Distributed systems can determine causality using logical clocks.

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. A process's actions are modeled as three types of events: internal, message send, and message receive. An internal event affects only the process at which it occurs, and the events at a process are linearly ordered by their order of occurrence. Send and receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process. Consequently, the execution of a distributed application results in a set of distributed events produced by the process. The causal precedence relation induces a partial order on the events of a distributed computation.

Causality among events, more formally the causal precedence relation, is a powerful concept for reasoning, analyzing, and drawing inferences about a distributed computation. The knowledege of the causal precedence relation between processes helps programmers, designers, and the system itself solve a variety of problems in distributed computing. In distributed algorithms design, such knowledge helps ensure liveness and fairness in mutual exclusion algorithms, maintains consistency in replicated databases, and helps design deadlock-detection algorithms that avoid phantom and undetected deadlocks. It also helps construct a consistent state for resuming reexecution in distributed debugging, build a checkpoint in failure recovery, and detect file inconsistencies in replicated databases. Such knowledge lets a process measure the progress of other processes, which is useful when discarding obsolete information, collecting garbage, and detecting termination. Finally, knowing the number of causally dependent events helps measure the amount of concurrency in a computation, since all events not causally related can be executed concurrently.

Human beings use the concept of causality to plan, schedule, and execute an enterprise, or to determine a plan's feasibility. In daily life, we use global time to deduce causality from loosely synchronized clocks such as wrist watches and wall clocks. But in distributed computing systems, the rate of event occurrence is several magnitudes higher, and the event-execution time several magnitudes smaller. If the physical clocks in these systems are not synchronized precisely, the causality relation between events cannot be captured accurately. The notion of time is basic to capturing the causality between events. However, distributed systems have no built-in physical time and can only approximate it. Even the Internet's Network Time Protocols,[1] which maintain a time accurate to a few tens of milliseconds, are not adequate for capturing causality in distributed systems. However, in a distributed computation, both the progress and the interaction between processes occurs in spurts. Consequently, we can use logical clocks to accurately capture the causality relation between events.

This article presents a general framework of a system of logical clocks in distributed systems and discusses three methods—scalar, vector, and

0018-9162/96/$5.00 © 1996 IEEE

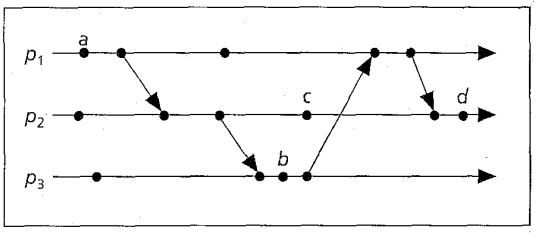

Figure 1. The time diagram of a distributed execution.

matrix—for implementing logical time in these systems. In these methods, time is represented by non-negative integers, a vector of non-negative integers, and a matrix of non-negative integers, respectively.

A MODEL OF DISTRIBUTED EXECUTIONS

A distributed program is composed of a set of n asynchronous processes $p_1, p_2, \ldots, p_i, \ldots, p_n$ that communicate by message-passing over a communication network. The processes do not share 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 they can access instantaneously. Process execution and message transfer are asynchronous. A process can execute an event spontaneously; when sending a message, it does not have to wait for the delivery to be complete.

Distributed executions

The execution of process p_i produces a sequence of events

$$e_i^0, e_i^1, \ldots, e_i^x, e_i^{x+1}, \ldots,$$

denoted by $\mathcal{H}_i$, where

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

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

We define the relation $\rightarrow_{msg}$ as follows: For every message m exchanged between two processes, we have

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

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

The distributed execution of a set of processes is a partial order $H = (H, \rightarrow)$, where $H = \cup_i h_i$ and $\rightarrow = (\cup_i \rightarrow_i \cup \rightarrow_{msg})^+$. The relation $\rightarrow$ expresses causal dependencies among the events in the distributed execution of a set of processes. If $e_1 \rightarrow e_2$, e_2 is directly or transitively dependent on e_1. If $e_2 \not\rightarrow e_1$ and $e_2 \not\rightarrow e_1$, events e_1 and e_2 are concurrent, denoted as $e_1 \,||\, e_2$. Clearly, for any two events e_1 and e_2 in a distributed execution, $e_1 \rightarrow e_2$, $e_2 \rightarrow e_1$, or $e_1 \,||\, e_2$.

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, and a slanted arrow indicates a message transfer. In this execution, $a \rightarrow b$, $b \rightarrow d$, and $b \,||\, c$.

Relevant events

Generally, few events are relevant at an observation or application level. For example, in a checkpointing protocol, only local checkpoint events are relevant. Let R denote the set of relevant events. Let $\rightarrow_R$ be the restriction of $\rightarrow$ to the events in R. That is,

$$\forall e_1, e_2 \in R: e_1 \rightarrow_R e_2 \Leftrightarrow e_1 \rightarrow e_2$$

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 ($R = \{a, b, c, d\}$), then $\rightarrow_R$ is defined as follows: $\rightarrow_R = \{(a, b), (a, c), (a, d), (b, d), (c, d)\}$.

LOGICAL CLOCKS: A MECHANISM TO CAPTURE CAUSALITY

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, by which a process can infer the causality relation between events. The timestamps assigned to events obey the fundamental monotonicity property. That is, if an event a causally affects an event b, the timestamp of a is smaller than the timestamp of b.

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 analogous to 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, denoted as $C(e)$ and called the timestamp of e, in the time domain T. The clock is defined as

$$C: H \mapsto T$$

to satisfy the following property:

$$e_1 \rightarrow e_2 \Rightarrow C(e_1) < C(e_2)$$

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

$$e_1 \rightarrow e_2 \Leftrightarrow C(e_1) < C(e_2)$$

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

Implementing logical clocks

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

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

- A local logical clock, denoted by lc_i, that helps p_i measure its own progress; and

- A global logical clock, denoted by gc_i, that represents p_i's local view of the global logical time. It allows the 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 governs how a process updates the local logical clock (to capture its progress) when it executes an event, whether send, receive, or internal.
- R2. This 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 a process piggybacks in a message and how the receiving process uses this information to update its view of the global time.

Systems of logical clocks differ in their representation of logical time and in the protocol for updating logical clocks. However, all logical clock systems implement some form of R1 and R2 and consequently ensure the fundamental monotonicity property associated with causality. Moreover, each logical clock system provides users with additional properties, as we discuss.

SCALAR TIME

Lamport proposed the scalar time representation in 1978[2] for totally ordering events in a distributed system. In this representation, the time domain 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 update the clocks as follows.

- R1. Before executing an event (send, receive, or internal), p_i executes the following:

$$C_i := C_i + d \qquad (d > 0)$$

In general, every time R1 is executed, d can have a different value, which can be application-dependent. However, d is typically kept at 1, since this allows a process to identify the time of each event uniquely at a process while minimizing d's rate of increase.

- R2. Each message piggybacks the clock value of its sender at sending time. When p_i receives a message with the timestamp C_{msg}, it executes the following actions:

1. $C_i := \max(C_i, C_{msg})$
2. Execute R1.
3. Deliver the message.

Figure 2 shows the evolution of scalar time, using $d = 1$ for the computation from Figure 1.

Basic properties

Clearly, scalar clocks satisfy monotonicity, and hence the consistency property. In addition, a distributed system

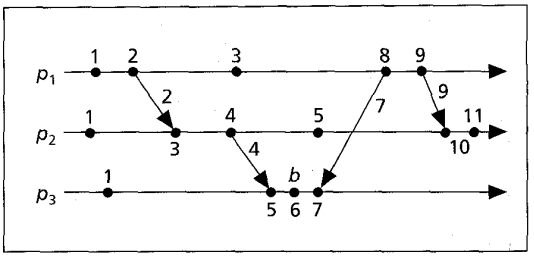

Figure 2. Evolution of scalar time in distributed execution.

can use scalar clocks to totally order events.[2] The main problem in totally ordering the events is that two or more events at different processes can have the identical timestamp. For example, in Figure 2, the third event of process p_1 and the second one of process p_2 receive the same scalar timestamp. We require a tie-breaking mechanism to order such events. Typically, process identifiers are linearly ordered, and a tie among events with the identical scalar timestamp is broken on the basis of their process identifiers. The timestamp of an event is denoted by a tuple (t, i), where t is its time of occurrence and i is the process at which it occurred. The total order relation $\prec$ on two events x and y with timestamps (h, i) and (k, j), respectively, is

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

Since events that occur at the same logical scalar time are independent (that is, not causally related), the system can order them using any criterion without violating the causality relation $\rightarrow$. Therefore, a total order is consistent with the causality relation $\rightarrow$. A 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).[2]

When the increment value d is always 1, scalar time has an interesting property. If event e has a timestamp h, then $h - 1$ represents the minimum logical duration, counted in events, required before producing e.[3] We call this the *height* of e. In other words, we know that $h - 1$ events have been produced sequentially before 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, the system of scalar clocks is not strongly consistent. That is, for two events e_1 and e_2,

$$C(e_1) < C(e_2) \not\Rightarrow e_1 \rightarrow e_2$$

For example, in Figure 2, the third event of process p_1 has a smaller scalar timestamp than the third event of p_2. However, the former did not happen before the latter. Scalar clocks are not strongly consistent because the local logical clock and global logical clock are squashed into one, losing the causal dependency information among events at different processes. In Figure 2, when p_2 receives the first message from p_1, it updates its clock to 3, forgetting that the timestamp of the latest event at p_1, on which it depends, is 2.

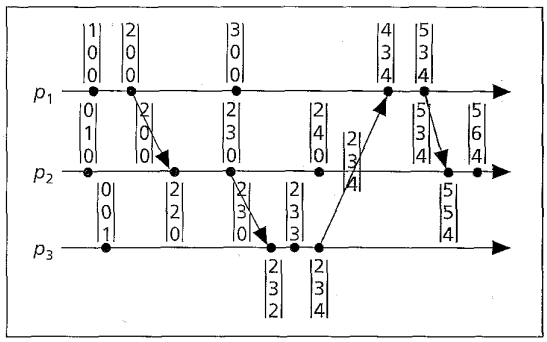

Figure 3. Evolution of vector time in distributed systems.

VECTOR TIME

Fidge,[3] Mattern,[4] and Schmuck[5] each developed a system of vector clocks independently (see "Vector clocks: A historical perspective" sidebar). 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 p_i. $vt_i[j]$ represents p_i's latest knowledge of p_j's local time. If $vt_i[j] = x$, p_i knows that the local time at p_j has progressed up to x. The entire vector vt_i constitutes p_i's view of the logical global time; p_i uses it to timestamp events.

The process p_i uses the following R1 and R2 to update its clock.

- R1. Before executing an event, p_i updates its local logical time as follows:

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

- R2. Each sender process piggybacks a message m with its vector clock value at sending time. Upon receiving such a message (m, vt), p_i executes the following sequence of actions:

  1. Update its logical global time as follows:

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

  2. Execute R1.
  3. Deliver the message m.

An event's timestamp is the value of its process's vector clock at the time the event is executed. Figure 3 shows an example of a vector clock's progression with the increment value $d = 1$.

Basic properties

ISOMORPHISM. The following three relations compare two vector timestamps, vh and vk:

$$\begin{aligned} vh \le vk &\Leftrightarrow \forall x : vh[x] \le vk[x] \\ vh < vk &\Leftrightarrow vh \le vk \textbf{ and } \exists x : vh[x] < vk[x] \\ vh \,||\, vk &\Leftrightarrow \textbf{not } (vh < vk) \textbf{ and not } (vk < vh) \end{aligned}$$

Recall that relation $\rightarrow$ induces a partial order on the set of events produced by a distributed execution. Timestamping events in a distributed system using a system of vector clocks creates the following property. If two events x and y have timestamps vh and vk, respectively, then:

$$\begin{aligned} x \rightarrow y &\Leftrightarrow vh < vk \\ x \,||\, y &\Leftrightarrow vh \,||\, vk \end{aligned}$$

An isomorphism thus exists between the set of partially ordered events produced by a distributed computation and their timestamps. This is a powerful, useful, and interesting property of vector clocks. If we know the process at which an event occurred, we can simplify the test to compare two timestamps as follows: If events x and y occurred respectively at processes p_i and p_j and are assigned timestamps (vh, i) and (vk, j) respectively,

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

STRONG CONSISTENCY. The system of vector clocks is strongly consistent. We can thus determine whether two events are causally related by comparing their vector timestamps. However, the dimension of vector clocks cannot be less than n for this property to apply.[6]

Vector clocks: A historical perspective

Although the theory associated with vector clocks was first developed in 1988, vector clocks were informally introduced and used by several researchers earlier. Parker et al.[1] used a rudimentary vector clocks system to detect inconsistencies of replicated files caused by network partitioning. Liskov and Ladin[2] proposed a vector clock system to define highly available distributed services. Strom and Yemini[3] used a similar system of clocks to track the causal dependencies between events in their optimistic recovery algorithm. Raynal[4] used them to prevent drift between logical clocks. Singhal[5] used vector clocks coupled with a Boolean vector to determine the currency of a critical section execution request by detecting the causality relation between a critical section request and its execution.

References

1. D.S. Parker et al., "Detection of Mutual Inconsistency in Distributed Systems," *IEEE Trans. Software Eng.*, Vol. SE-9, No. 3, May 1983, pp. 240-246.
2. B. Liskov and R. Ladin, "Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection," *Proc. 5th ACM Symp. Principles Distributed Computing*, ACM Press, New York, 1986, pp. 29-39.
3. R.E. Strom and S. Yemini, "Optimistic Recovery in Distributed Systems," *ACM Trans. Computer Systems*, Vol. 3, No. 3, Aug. 1985, pp. 204-226.
4. M. Raynal, "A Distributed Algorithm to Prevent Mutual Drift Between n Logical Clocks," *Information Processing Letters*, Vol. 24, 1987, pp. 199-202.
5. M. Singhal, "A Heuristically-Aided Mutual Exclusion Algorithm for Distributed Systems," *IEEE Trans. Computers*, Vol. 38, No. 5, May 1989, pp. 651-662.

EVENT COUNTING. If d is always 1 in the rule R1, the ith component of vector clock at 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 the timestamp vh, $vh[j]$ denotes the number of events executed by 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, it is used in distributed debugging, implementing causal ordering communication and causal distributed shared memory, establishing global breakpoints, and implementing the consistency of checkpoints in optimistic recovery.

MATRIX TIME

Michael and Fischer informally proposed a system of matrix clocks in 1982.[7] Both Wuu and Bernstein[8] and Lynch and Sarin[9] employed the system to discard obsolete information in replicated databases. In a system of matrix clocks, time is represented by a set of $n \times n$ matrices of nonnegative 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 p_i;
- $mt_i[i, j]$ denotes the latest knowledge that p_i has about the local logical clock, $mt_j[j, j]$, of p_j (note that row $mt_i[i, .]$ is nothing but the vector clock $vt_i[.]$ and exhibits all the properties of vector clocks); and
- $mt_i[j, k]$ represents what p_i knows about the latest knowledge that p_j has about the local logical clock, $mt_k[k, k]$, of p_k.

The entire matrix mt_i denotes p_i's local view of the logical global time. The matrix timestamp of an event is the value of the matrix clock of the process when the event is executed.

Process p_i uses the following rules R1 and R2 to update its clock. According to R1, before executing an event, p_i updates its local logical time as follows:

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

Under R2, each message m is piggybacked with the matrix time mt. When p_i receives such a message (m, mt) from p_j, p_i executes the following sequence of actions:

1. Update its logical global time as follows:

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

2. Execute R1.
3. Deliver message m.

Figure 4 shows how matrix clocks progress in a distributed computation. We assume $d = 1$, so every event at a process gets a locally unique sequence number. Let us consider the following events: e, which is the x_ith event at process p_i; e_k^1 and e_k^2, which are the x_k^1th and x_k^2th event at process p_k; and e_j^1 and e_j^2, which are the x_j^1th and x_j^2th events

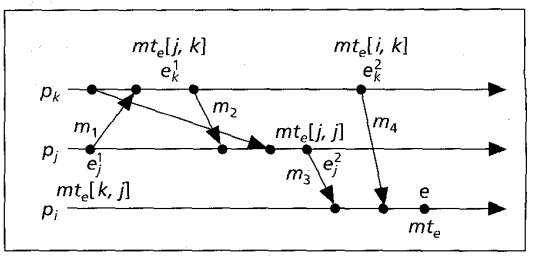

Figure 4. Evolution of matrix time in distributed systems.

at p_j. Let mt_e denote the matrix timestamp associated with e. Due to message m_4, e_k^2 is the last event of p_k that causally precedes e, therefore, $mt_e[i,k] = mt_e[k,k] = x_k^2$. Likewise, $mt_e[i,j] = mt_e[j,j] = x_j^2$. The last event of p_k known by p_j, as far as p_i knew when it executed e, is e_k^1; therefore, $mt_e[j,k] = x_k^1$. Likewise, we have $mt_e[k,j] = x_j^1$.

Basic properties

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

$$\min_k \left(mt_i[k,l]\right) \ge t \Rightarrow$$ process p_i knows that every other process p_k knows the p_l's local time has progressed until t

If this is true, p_i knows that all other processes know that p_l will never send information with a local time $\le t$. In many applications, this implies that processes will no longer require certain information from p_l 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 knows.

EFFICIENT IMPLEMENTATIONS

When there are a large number of processes in a distributed computation, the vector and matrix clocks must piggyback huge amounts of information in messages to disseminate time progress and update the clocks. In this section, we discuss efficient ways to maintain vector clocks; we could use similar techniques to efficiently implement matrix clocks.

If vector clocks must satisfy the strong consistency property, vector timestamps must be at least of size n.[6] Therefore, in general, the size of a vector timestamp equals the number of processes involved in a distributed computation. However, several optimizations are possible.

Singhal-Kshemkalyani's differential technique

Singhal and Kshemkalyani's technique[10] is based on an observation that between successive events at a process, only a few entries of the vector clock are likely to change. This is more likely when the number of processes is large, since only a 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, p_i piggybacks only those entries of its vector clock that have changed since the last message it sent to p_j. Therefore,

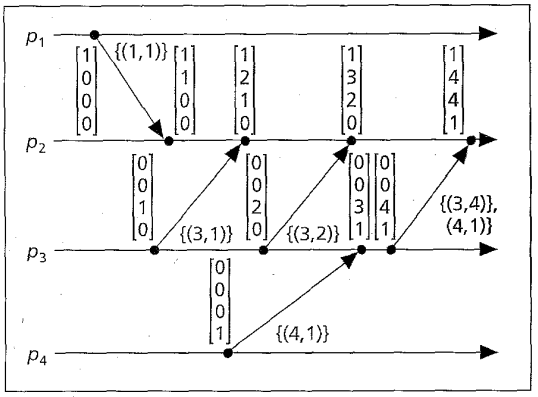

Figure 5. The Singhal-Kshemkalyani technique for vector clocks.

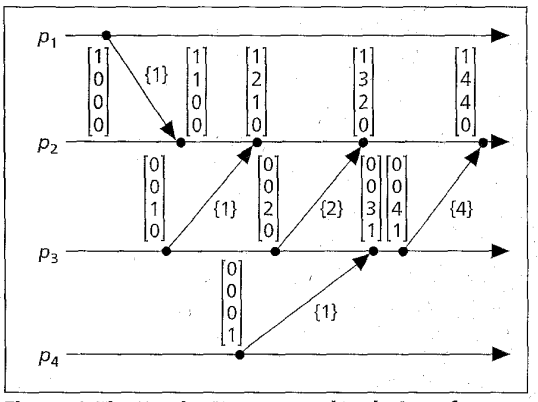

Figure 6. The Fowler-Zwaenepoel technique for vector clocks.

this technique cuts down the communication bandwidth and buffer requirements (to store messages). However, a process needs to maintain two additional vectors to store the information regarding the clock values at the time of the last interaction with other processes.

Figure 5 illustrates the Singhal-Kshemkalyani technique. If entries $i_1, i_2, \ldots, i_{n1}$ of p_i's vector clock have changed (to $v_1, v_2, \ldots, v_{n1}$, respectively) since the last message to p_j, p_i piggybacks a compressed timestamp $\{(i_1, v_1), (i_2, v_2), \ldots, (i_{n1}, v_{n1})\}$ in its 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 $k = 1, 2, \ldots, n1$. 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 first-in, first-out.

Fowler-Zwaenepoel's direct-dependency technique

Fowler-Zwaenepoel's direct-dependency technique[11] does not maintain vector clocks on the fly. Instead, a process maintains information regarding only direct dependencies on other processes. It constructs a vector time for an event, representing transitive dependencies on other processes, 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 $j = 1 .. n$. p_i updates it as follows:

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

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

Figure 6 illustrates the Fowler-Zwaenepoel technique. The technique provides considerable cost savings, since only one scalar is piggybacked on every message. However, the dependency vector does not represent transitive dependencies (that is, vector timestamps). Instead, the technique obtains the transitive dependency of an event by recursively tracing the direct-dependency vectors of processes. This will obviously create overhead and latencies, making the technique unsuitable for applications that require on-the-fly computation of vector timestamps. Nonetheless, it is ideal for applications that compute causal dependencies off line, such as causal breakpoint and asynchronous checkpointing recovery.

Jard-Jourdan's adaptive technique

The Fowler-Zwaenepoel technique requires a process to observe an event—that is, update and record its dependency vector—after receiving a message and before sending out any. Otherwise, reconstructing a vector timestamp from the direct-dependency vectors will not capture all causal dependencies. When events are highly frequent, this technique requires recording the history of a large number of events. The Jard-Jourdan technique[12] lets processes adaptively observe events while maintaining the ability to retrieve all the causal dependencies of such events.

Jard and Jourdan defined the pseudodirect relation $<<$ on the events of a distributed computation as follows. If events e_i and e_j occur at processes p_i and p_j, respectively, then $e_j << e_i$ if and only if a path of message transfers exists which starts after e_j on p_j and ends before e_i on p_i, such that no observed event exists on the path.

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

Whenever an event is observed at p_i, the following actions are executed, (let $p\_vt_i = \{(i1,v1), \ldots, (i,v), \ldots\}$ denote the current partial vector clock at p_i and variable e_vt_i holds the timestamp of the observed event):

- $e\_vt_i = \{(i1, v1), \ldots, (i,v), \ldots\}$
- $p\_vt_i := \{(i, v+1)\}$

When p_j sends a message to p_i, it piggybacks the current value of p_vt_j in the message. When p_i receives a message piggybacked with the timestamp p_vt, p_i sets p_vt_i to the union of the following (let $p\_vt = \{(i_{m1}, v_{m1}), \ldots, (i_{mk}, v_{mk})\}$

and $p\_vt_i = \{(i_1, v_1), \ldots, (i_l, v_l)\}$):

- 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 7 illustrates the Jard-Jourdan technique for maintaining vector clocks. eX_pt_n denotes the timestamp of the Xth observed event at p_n. For example, the third event observed at p_3 is timestamped $e3\_pt_3 = \{(3, 2), (4, 1)\}$. This timestamp means that the pseudodirect predecessors of this event are respectively the second event observed at p_3 and the first observed at p_4. So, given the timestamp of an event, we can easily compute the set of observed events that are its predecessors.

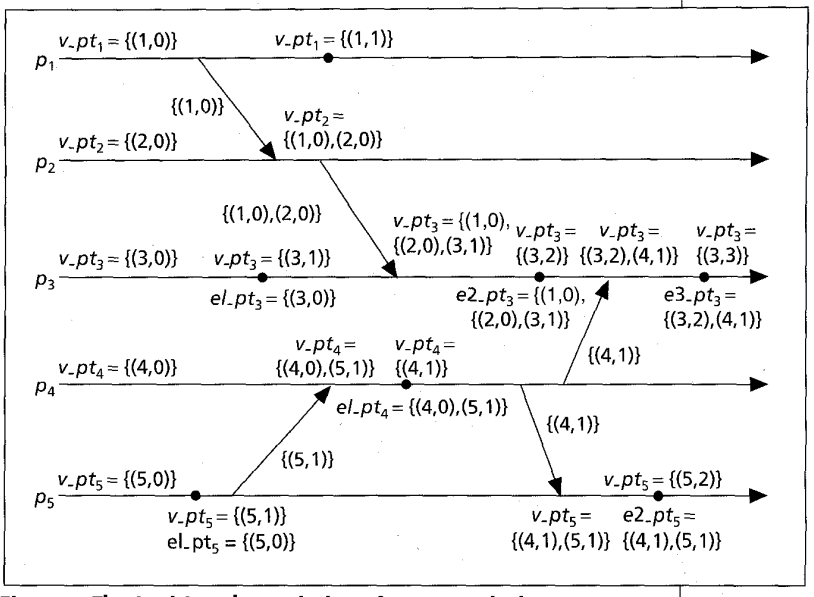

Figure 7. The Jard-Jourdan technique for vector clocks.

THE CONCEPT OF CAUSALITY AMONG EVENTS is fundamental to the design and analysis of distributed programs. The notion of time is basic to capturing causality between events; however, distributed systems can only realize an approximation of time. Because a distributed computation typically progresses in spurts, logical time, which advances in jumps, can capture the monotonicity property induced by causality in the system. Causality among events in a distributed system is a powerful concept in reasoning, analyzing, and drawing inferences about a computation. Another notion of global time that preexists in the semantics of distributed programs and participates in the execution of such programs is called virtual time (see the "Virtual time" sidebar).

Virtual time

Awerbuch's *synchronizer* concept[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; its progress relies on a global time assumption. In the semantics of synchronous distributed programs, a global time preexists and participates in the execution of such programs. A synchronizer interprets synchronous distributed programs and simulates a global time for them in an asynchronous environment.[2]

In distributed, discrete-event simulations,[3,4] the semantics of the simulation program rely on a global (or so-called simulation) time. 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 (has the liveness property) in a way that avoids violating the causality relations of the program, providing the necessary safety conditions.

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. It belongs to the underlying program semantics and is nothing but the virtual[5] counterpart of the physical time offered by the environment and used in real-time applications. On the other hand, logical time (whether linear, vector, or matrix) orders events according to their causal precedence to ensure properties such as liveness, consistency, and fairness. Such logical time is just one means to ensure these properties. Ricart-Agrawala's mutual exclusion algorithm[6] uses Lamport's logical clocks to ensure liveness; this time belongs neither to the mutual exclusion semantics nor the program invoking mutual exclusion. In fact, other means can ensure properties such as liveness. For example, Chandy and Misra's mutual exclusion algorithm[7] employs a dynamic, directed, acyclic graph instead of clocks to ensure liveness.

References

1. B. Awerbuch, "Complexity of Network Synchronization," *J. ACM*, Vol. 32, No. 4, 1985, pp. 804-823.
2. M. Raynal and J.M. Helary, *Synchronization and Control of Distributed Systems and Programs*, John Wiley & Sons, New York, 1990, 124 pp.
3. J. Misra, "Distributed Discrete Event Simulation," *ACM Computing Surveys*, Vol. 18, No. 1, 1986, pp. 39-65.
4. R. Righter and J.C. Walrand, "Distributed Simulation of Discrete Event Systems," *Proc. IEEE*, Jan. 1988, pp. 99-113.
5. D. Jefferson, "Virtual Time," *ACM Trans. Programming Languages and Systems*, Vol. 7, No. 3, 1985, pp. 404-425.
6. G. Ricart and A.K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Comm. ACM*, Vol. 24, No. 1, Jan. 1981, pp. 9-17.
7. K.M. Chandy and J. Misra, "The Drinking Philosophers Problem," *ACM Trans. Programming Languages and Systems*, Vol. 6, No. 4, 1984, pp. 632-646.

We have presented a general framework of logical clocks in distributed systems and have discussed three systems of logical clocks: scalar, vector, and matrix. These systems have been used to solve a variety of problems in distributed algorithm design, debugging distributed programs, checkpointing and failure recovery, data consistency in replicated databases, discarding obsolete information, garbage collection, and termination detection.

In scalar clocks, the clock at a process is represented by an integer. The message and computation overheads are small, but the power of scalar clocks is limited—they are not strongly consistent. In vector clocks, the clock at a process is represented by a vector of integers. Thus, the message and computation overheads are likely to be high; however, vector clocks possess a powerful property—the isomorphism that exists between the set of partially ordered events in a distributed computation and their vector timestamps. This useful, interesting property of vector clocks finds applications in several problem domains. In matrix clocks, the clock at a process is represented by a matrix of integers. Thus, the message and computation overheads are high; however, matrix clocks are quite powerful. Besides containing information about the direct dependencies, a matrix clock contains information about the latest direct dependencies of those dependencies, This information can be useful in applications such as distributed garbage collection. Thus, the power of systems of clocks increases in the order of scalar, vector, and matrix, but so do the complexity and overheads.

We discussed three efficient implementations of vector clocks; similar techniques can be used to efficiently implement matrix clocks. ■

Acknowledgments

We are deeply grateful to the four anonymous referees for their comments on a previous version of this article.

References

1. D.L. Mills, "On the Accuracy and Stability of Clocks Synchronized by Network Time Protocol in the Internet System," *ACM Computer Comm. Rev.*, Vol. 20, No. 1, Jan. 1990, pp. 65-75.
2. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, Vol. 21, No. 7, July 1978, pp. 558-564.
3. C. Fidge, "Logical Time in Distributed Computing Systems," *Computer*, Vol. 24, No. 8, Aug. 1991, pp. 28-33.
4. F. Mattern, "Virtual Time and Global States of Distributed Systems," *Proc. Parallel and Distributed Algorithms Conf.*, North-Holland, Amsterdam, 1988, pp. 215-226.
5. F. Schmuck, *The Use of Efficient Broadcast in Asynchronous Distributed Systems*, doctoral dissertation, Tech. Report TR88-928, Dept. Computer Science, Cornell Univ., Ithaca, New York, 1988, 124 pp.
6. B. Charron-Bost, "Concerning the Size of Logical Clocks in Distributed Systems," *Information Processing Letters*, Vol. 39, July 1991, pp. 11-16.
7. M.J. Fischer and A. Michael, "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network," *Proc. ACM Symp. Principles Database Systems*, ACM Press, New York, 1982, pp. 70-75.
8. G.T.J. Wuu and A.J. Bernstein, "Efficient Solutions to the Replicated Log and Dictionary Problems," *Proc. 3rd ACM Symp. Principles Distributed Computing*, (PODC), ACM Press, New York, 1984, pp. 233-242.
9. S.K. Sarin and L. Lynch, "Discarding Obsolete Information in a Replicated Data Base System," *IEEE Trans. Software Eng.*, Vol. SE, No. 13.1, Jan. 1987, pp. 39-46.
10. M. Singhal and A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," *Information Processing Letters*, Vol. 43, Aug. 1992, pp. 47-52.
11. J. Fowler and W. Zwaenepoel, "Causal Distributed Breakpoints," *Proc. 10th Int'l Conf. Distributed Computing Systems*, 1990, pp. 134-141.
12. C. Jard and G-C. Jourdan, "Dependency Tracking and Filtering in Distributed Computations," in *Brief Announcements ACM Symp. Principles Distributed Computing*, ACM Press, New York, 1994; also Tech. Report No. 851, IRISA, Beaulieu, France, 1994.

Michel Raynal *is a professor of computer science at the University of Rennes, France. His research interests are distributed algorithms, operating systems, protocols and parallelism. He received the Doctorat d'Etat en informatique in 1981 from Rennes University. He has written seven books devoted to distributed computing systems, including Distributed Computations and Networks (MIT Press, 1988) and Synchronization and Control of Distributed Programs (Wiley & Sons, 1990). He chaired the 9th International Workshop on Distributed Algorithms (WDAG9) in France. He is currently involved in European Esprit projects devoted to the design of fault-tolerant distributed systems.*

Mukesh Singhal *is an associate professor of computer and information science at Ohio State University, Columbus. His current research interests include distributed systems, operating systems, mobile computing, and performance modeling. He received a B.Eng. in electronics and communication engineering from the University of Roorkee, Roorkee, India, in 1980 and a PhD in computer science from the University of Maryland, College Park, in 1986. He coauthored* Advanced Concepts in Operating Systems *(McGraw-Hill, 1994). He is an editor of the IEEE Computer Society Press.*

Readers can contact Raynal at IRISA, Campus de Beaulieu, 35042 Rennes-Cédex, France, e-mail raynal@irisa.fr. Singhal can be reached at the Dept. of Computer and Information Science, Ohio State University, Columbus, OH 43210, e-mail singhal@cis.ohio-state.edu.

Doris Carver, Computer's *Software Technologies area editor, coordinated the review of this article and recommended it for publication. Her e-mail address is carver@bit.csc.lsu.edu.*