Logical Time:

Capturing Causality in Distributed Systems

Michel Raynal
University of Rennes

Mukesh Singhal
Ohio State University

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 knowledge 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, 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 spurs. 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
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 $p_1, p_2, \ldots, p_n$, 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 $e_i^0, e_i^1, \ldots, e_i^n, \ldots,$ denoted by $\mathcal{E}_i$ where

$$\mathcal{E}_i = (h_i \rightarrow)$$

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

We define the relation $\rightarrow_{\text{seg}}$ as follows: For every message $m$ exchanged between two processes, we have $\text{send}(m) \rightarrow_{\text{seg}} \text{receive}(m)$.

The relation $\rightarrow_{\text{seg}}$ 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 $H = \bigcup_{i=1}^{n} h_i$ and $\rightarrow = (\cup_{i=1}^{n} \rightarrow)$. The relation $\rightarrow$ expresses causal dependencies among the events in the distributed execution of a set of processes. If $e_1 \rightarrow e_2, e_2 \rightarrow e_3$ is directly or transitively dependent on $e_1$. If $e_2 \rightarrow e_1$ and $e_2 \rightarrow e_1$, events $e_1$ and $e_2$ are concurrent, denoted as $e_1 \parallel e_2$. Clearly, for any two events $e_1$ and $e_2$ in a distributed execution, $e_1 \rightarrow e_2 \Rightarrow e_2 \rightarrow e_1$, 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, and a slanted arrow indicates a message transfer. In this execution, $a \rightarrow b, b \rightarrow d$, and $b \parallel c$.

**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 $R_{\rightarrow}$ be the restriction of $\rightarrow$ to the events in $R$. That is,

$$\forall e_1, e_2 \in R: e_1 \rightarrow_{R} e_2 \Rightarrow e_1 \rightarrow e_2$$

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 $\mathcal{P} = (R, \rightarrow)$. For example, if in Figure 1, only events $a, b, c, d$ are relevant to an observation level ($R = \{(a, b, c, d),\}$), then $R_{\rightarrow}$ is defined as follows: $R_{\rightarrow} = \{(a, b), (a, c), (a, d), (b, d), (c, d)\}$.

**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 $e_2$ causally affects an event $e_1$, the timestamp of $e_2$ is no smaller than the timestamp of $e_1$.

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

$$C : H \rightarrow T$$

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,

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

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 $t_{\text{local}}$, that helps $p_i$ measure its own progress; and
• A global logical clock, denoted by \( g_c \), 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, \( h \) is a part of \( g_c \).

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 local 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:

\[
C_i := C_i + d \quad (d > 0)
\]

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 \( C_{\text{from}} \), it executes the following actions:

1. \( C_i := \max(C_i, C_{\text{from}}) \)
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 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, event identifiers are linearly ordered, and a tie among events with the identical scalar timestamp is broken on the basis of their event identifiers. The timestamp of an event is denoted by a tuple \((r, i)\), where \( r \) 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

\[
x \prec y \iff \left( h < k \text{ or } (h = k \text{ and } i < j) \right)
\]

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 10 on the longest causal path ending at 10.

However, the system of scalar clocks is not strongly consistent. That is, for two events \( e_1 \) and \( e_2 \),

\[
C(e_1) < C(e_2) \Rightarrow e_1 \rightarrow e_2
\]

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 \( p_3 \) 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.

February 1996
VECTOR TIME

Fidge, Matern, and Schmuck 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 maintains a vector \( v_t[i] \), where \( v_t[i] \) is the local logical clock of \( p_i \) and describes the logical time progress at \( p_i \). \( v_t[i] \) represents \( p_i \)'s latest knowledge of \( p_i \)'s local time. If \( v_t[i] = x, p_i \) knows that the local time at \( p_i \) has progressed up to \( x \). The entire vector \( v_t \) constitutes \( p_i \)'s view of the logical global time; \( p_i \) uses it to timestamp events.

The process \( p_1 \) uses the following R1 and R2 to update its clock:

- R1. Before executing an event, \( p_i \) updates its local logical time as follows:
  \[
  v_t[i] := v_t[i] + d \quad (d > 0)
  \]
- R2. Each sender process piggybacks a message \( m \) with its vector clock value at sending time. Upon receiving such a message \((m, v_t)\), \( p_i \) executes the following sequence of actions:
  1. Update its logical global time as follows:
     \[
     1 \leq k \leq n : v_t[k] := \max(v_t[k], v[k])
     \]
  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, \( v_h \) and \( v_k \):

\[
\begin{align*}
\forall x : v_h(x) &\leq v_k(x) \\
v_h < v_k &\iff v_h \leq v_k \text{ and } \exists x : v_h(x) < v_k(x) \\
v_h || v_k &\iff \text{not } (v_h < v_k) \text{ and not } (v_k < v_h)
\end{align*}
\]

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 \( v_h \) and \( v_k \), respectively, then:

\[
\begin{align*}
x \rightarrow y &\iff v_h < v_k \\
x || y &\iff v_h || v_k
\end{align*}
\]

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 \( v_h(i) \) and \( v_k(j) \), respectively,

\[
\begin{align*}
x \rightarrow y &\iff v_h[i] < v_k[j] \\
x || y &\iff v_h[i] > v_k[j] \text{ and } v_h[j] < v_k[i]
\end{align*}
\]

**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.
EVENT COUNTING. If \( d \) is always 1 in the rule R1, the \( i \)th component of vector clock at \( p_i \), \( vt[i] \), denotes the number of events that have occurred at \( p_i \) until that instant. So if an event \( e \) has the timestamp \( vh[j] \), \( vh[j] \) denotes the number of events executed by \( p_i \) that causally preceded \( e \). Clearly, \( \sum vh[i] - 1 \) 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. Both Wu and Bernstein and Lynch and Sarin 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 non-negative integers. A process \( p_i \) maintains a matrix \( mt[i][j] \), for \( 1 \leq i, j \leq n \), where

- \( mt[i][j] \) denotes the local logical clock of \( p_i \) and tracks the progress of the computation at \( p_i \);
- \( mt[i][j] \) denotes the latest knowledge that \( p_i \) has about the local logical clock, \( mt[j][i] \), of \( p_j \). (Note that row \( mt[i][] \) is nothing but the vector clock \( vt[i] \) and exhibits all the properties of vector clocks); and
- \( mt[i][k] \) represents what \( p_i \) knows about the latest knowledge that \( p_i \) has about the local logical clock, \( mt[k][k] \), of \( p_k \).

The entire matrix \( mt \) denotes \( p_i \)'s local view of the logical global time. The matrix timestamp of an event is the value of the matrix of the clock at the moment 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 logical time as follows:

\[
mt[i][j] := mt[i][j] + d \quad (d > 0)
\]

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

1. Update its logical global time as follows:
   \[
   1 \leq k \leq n : \text{if} \ mt[i][k] = \max(mt[i][k], mt[j][k])
   \]
   \[
   1 \leq k, l \leq n : \text{if} \ mt[k][l] = \max(mt[k][l], mt[k][l])
   \]
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 \( x \)th event at process \( p_x \) and \( e_j \) which is the \( x \)th event at process \( p_x \) and \( e_j \) which is the \( x \)th event at process \( p_x \); and \( e_j \) and \( e_j \), which are the \( x \)th and \( x \)th events at \( p_x \). Let \( mt \), denote the matrix timestamp associated with \( e \). Due to message \( m_t \), \( e_j \) is the last event of \( p_k \) that causally precedes \( e \), therefore, \( mt[i][k] = mt[k][k] = x \). Likewise, \( mt[i][j] = \max(mt[j][j], x) \). The last event of \( p_j \) known by \( p_x \), as far as \( p_x \) knew when it executed \( e \), is \( e_j \); therefore, \( mt[j][k] = x \). Likewise, we have \( mt[k][j] = x \).

**Basic properties**

Clearly, the vector \( mt[i][] \) contains all the properties of vector clocks. In addition, matrix clocks have the following property:

\[
\min_k \left( \{mt[i][j]\} \right) \geq t \Rightarrow \text{process } p_i \text{ knows that every other process } p_j \text{ knows the } p_i \text{'s local time has progressed until } t
\]

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

If \( d \) is always 1 in the rule R1, then \( mt[k][] \) denotes the number of events occurred at \( p_k \) and known by \( p_x \) as far as \( p_x \) 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 \( n \). 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\(^6\) 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 \( p_x \) sends a message to a process \( p_x \), \( p_x \) piggybacks only those entries of its vector clock that have changed since the last message it sent to \( p_x \). Therefore,
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 \( t_1, t_2, \ldots, t_n \) of \( p_i \)'s vector clock have changed (to \( v_{12}, v_{13}, \ldots, v_{1n} \), respectively) since the last message to \( p_j \), \( p_j \) piggybacks a compressed timestamp \((t_1, v_{12}), (t_2, v_{13}), \ldots, (t_n, v_{1n})\) in its next message to \( p_j \). When \( p_j \) receives this message, it updates its clock as follows: \( v_{k+} = \max(v_k, v_{j1}) \) for \( k = 1, 2, \ldots, n \). 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\(^1\) 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 \( j = 1, \ldots, n \). \( p_i \) updates it as follows:

- When an event occurs at \( p_j \), \( D_i(j) = D_i(j) + 1 \).
- When \( p_j \) sends a message \( m \) to \( p_k \), \( p_k \) piggybacks the updated value of \( D_i \) in the message.
- When \( p_k \) receives a message from \( p_j \) with the piggybacked value \( d \), \( p_k \) 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 on \( p_j \) 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 tracking the direct-dependency vectors of processes. This will obviously increase overhead and latencies, making the technique unsuited 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\(^2\) allows 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 \( e_i << e_j \) if and only if a path of message transfers exists which starts after \( e_i \) on \( p_j \) and ends before \( e_j \) on \( p_i \) such that no observed event exists on the path.

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

Whenever an event is observed at \( p_i \), the following actions are executed, (let \( p_i.v = ((1, v_1), \ldots, (j, v_j), \ldots) \) denote the current partial vector clock at \( p_i \) and variable \( e_i.v \) holds the timestamp of the observed event):

- \( e_i.v = ((1, v_1), \ldots, (j, v_j), v_{j+1}) \)
- \( p_j.v = (j, v_j + 1) \)

When \( p_j \) sends a message to \( p_i \), it piggybacks the current value of \( p_j.v \) in the message. When \( p_j \) receives a message piggybacked with the timestamp \( p_i.v \), \( p_j \) updates \( p_i.v \) to the union of the following (let \( p_i.v = ((1, v_1), \ldots, (j, v_j), v_{j+1}) \))
and \( p \cdot \forall t_i = \{ (i, \nu_i), \ldots, (i, \nu_i) \} \):

- all \((i_{\text{min}}, \nu_i)\) such that \((i_{\text{min}}, .)\) does not appear in \(v \cdot pt\),
- all \((i, \nu_i)\) such that \((i, .)\) does not appear in \(v \cdot pt\), and
- all \(\{i_{\text{max}}(\nu_i, \nu_i)\}\) for all \((\nu_i, .)\) that appear in \(v \cdot pt\) and \(v \cdot pt\).

Figure 7 illustrates the Jard-Joudan technique for maintaining vector clocks. \(eX \cdot pt\), denotes the timestamp of the \(X\)th observed event at \(p_e\). For example, the third event observed at \(p_3\) is timestamped \(e3 \cdot pt_3 = \{(3, 2), (4, 1)\}\). This timestamp means that the pseudodirect predecessors of this event are respectively the second event observed at \(p_3\) and the first observed at \(p_3\). So, given the timestamp of an event, we can easily compute the set of observed events that are its predecessors.

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 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.

In distributed, discrete-event simulations, 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 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 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 employs a dynamic, directed, acyclic graph instead of clocks to ensure liveness.

References

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, 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. 1

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

References

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 (WDAG99) 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 Ph.D. 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-Cedex, France, e-mail raynal@iris.fr. Singhal can be reached at the Dept. of Computer and Information Science, Ohio State University, Columbus, OH 43210, e-mail singhal@cs.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.