CHAPTER

3 Logical time

3.1 Introduction

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. However, in distributed systems, it is not possible to have global physical time; it is possible to realize only 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 chapter discusses three ways to implement logical time (e.g., scalar time, vector time, and matrix time) that have been proposed to capture causality between 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 the events of processes helps solve a variety of problems in distributed systems. Examples of some of these problems is as follows:

- **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 measure the progress of 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 the planning, scheduling, and execution of a chore or an enterprise, or in determining the infeasibility of a plan or the innocence of an accused. In day-to-day 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. Network Time Protocols [15], which can maintain time accurate to a few tens of milliseconds on the Internet, are not adequate to capture the causality relation in distributed systems. However, in a distributed computation, generally the 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 chapter 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, 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. We also discuss methods for efficient implementation of the systems of vector clocks.

The chapter ends with a discussion of virtual time, its implementation using the time-warp mechanism and a brief discussion of physical clock synchronization and the Network Time Protocol.

3.2 A framework for a system of logical clocks

3.2.1 Definition

A system of logical clocks consists of a time domain T and a logical clock C [19]. Elements of T form a partially ordered set over a relation $<$. This relation is usually called the *happened before* or *causal precedence*. Intuitively, this relation is analogous to the *earlier than* relation provided by the 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 $\mathrm{C}(e)$ and called the timestamp of e, and is defined as follows:

$$C : H \mapsto T,$$

such that the following property is satisfied:

$$\text{for two events } e_i \text{ and } e_j,\ e_i \rightarrow e_j \Longrightarrow \mathrm{C}(e_i) < \mathrm{C}(e_j).$$

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

$$\text{for two events } e_i \text{ and } e_j,\ e_i \rightarrow e_j \Leftrightarrow \mathrm{C}(e_i) < \mathrm{C}(e_j),$$

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

3.2.2 Implementing logical clocks

Implementation of logical clocks requires addressing two issues [19]: 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 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 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 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.

3.3 Scalar time

3.3.1 Definition

The scalar time representation was proposed by Lamport in 1978 [9] 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:

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

  In general, every time **R1** is executed, d can have a different value, and this value may be application-dependent. However, typically d is kept at 1 because this is able to identify the time of each event uniquely at a process, while keeping the rate of increase of d to its lowest level.
- **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:
  1. $C_i := max(C_i, C_{msg})$;
  2. execute **R1**;
  3. deliver the message.

Figure 3.1 shows the evolution of scalar time with $d = 1$.

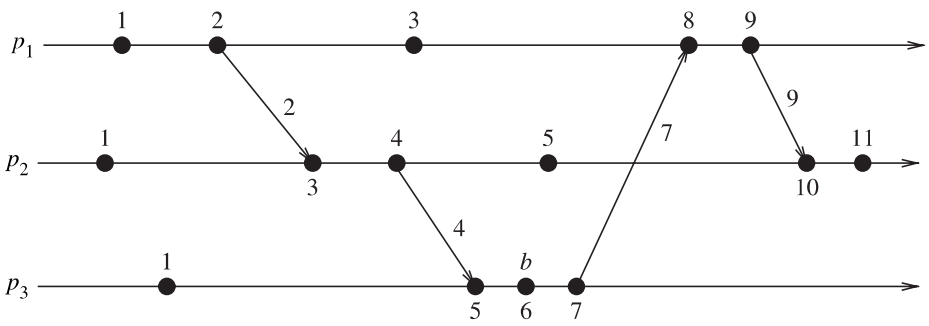

Figure 3.1 Evolution of scalar time [19].

3.3.2 Basic properties

Consistency property

Clearly, scalar clocks satisfy the monotonicity and hence the consistency property:

$$\text{for two events } e_i \text{ and } e_j,\ e_i \rightarrow e_j \Longrightarrow C(e_i) < C(e_j).$$

Total Ordering

Scalar clocks can be used to totally order events in a distributed system [9]. The main problem in totally ordering events is that two or more events at different processes may have an identical timestamp. (Note that for two events e_1 and e_2, $C(e_1) = C(e_2) \Longrightarrow e_1 \parallel e_2$.) For example, in Figure 3.1, the third event of process P_1 and the second event of process P_2 have identical scalar timestamp. Thus, a tie-breaking mechanism is needed to order such events. Typically, a tie is broken as follows: process identifiers are linearly ordered and a tie among events with identical scalar timestamp is broken on the basis of their process identifiers. The lower the process identifier in the ranking, the higher the priority. 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:

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

Since events that occur at the same logical scalar time are independent (i.e., they are not causally related), they can be ordered using any arbitrary criterion without violating the causality relation $\rightarrow$. Therefore, a total order is consistent with the causality relation "$\rightarrow$". Note that $x \prec y \Longrightarrow x \rightarrow y \vee x \parallel y$. A total order is generally used to ensure liveness properties in distributed algorithms. Requests are timestamped and served according to the total order based on these timestamps [9].

Event counting

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 [4]; we call it the height of the event 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 3.1, five events precede event b on the longest causal path ending at b.

No strong consistency

The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j, $C(e_i) < C(e_j) \not\Longrightarrow e_i \rightarrow e_j$. For example, in Figure 3.1, the third event

of process P_1 has smaller scalar timestamp than the third event of process P_2. However, the former did not happen before the latter. The reason that scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss causal dependency information among events at different processes. For example, in Figure 3.1, when process P_2 receives the first message from process 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.

3.4 Vector time

3.4.1 definition

The system of vector clocks was developed independently by Fidge [4], Mattern [12], and Schmuck [23]. 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 global logical 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, process p_i 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 executes the following sequence of actions:
  1. update its global logical time as follows:

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

  2. execute **R1**;
  3. 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 3.2 shows an example of vector clocks progress with the increment value $d = 1$. Initially, a vector clock is $[0, 0, 0, \ldots, 0]$.

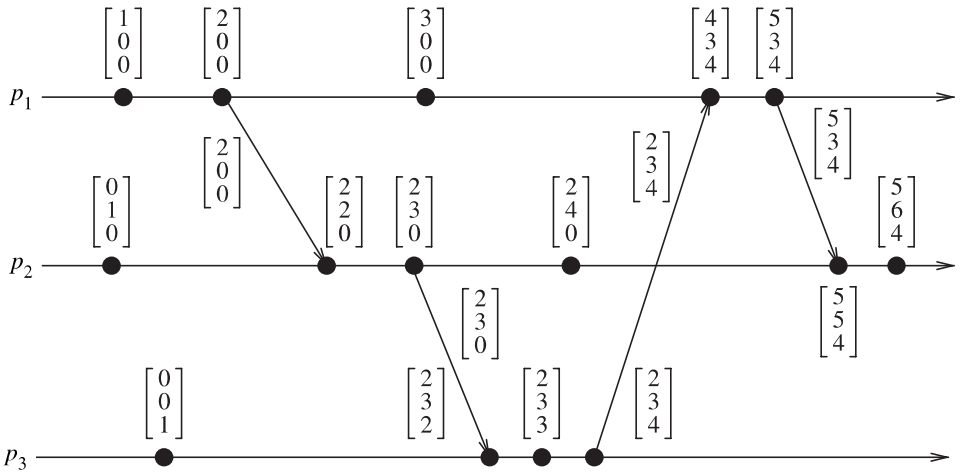

Figure 3.2 Evolution of vector time [19].

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

$$vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$$
$$vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$$
$$vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$$
$$vh \parallel vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh).$$

3.4.2 Basic properties

Isomorphism

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

$$x \rightarrow y \Leftrightarrow vh < vk$$
$$x \parallel y \Leftrightarrow vh \parallel vk.$$

Thus, there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps. This is a very powerful, useful, and interesting property of vector clocks.

If the process at which an event occurred is 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 and vk, respectively, then

$$x \rightarrow y \Leftrightarrow vh[i] \leq vk[i]$$
$$x \parallel y \Leftrightarrow vh[i] > vk[i] \wedge vh[j] < vk[j].$$

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, Charron–Bost showed that the dimension of vector clocks cannot be less than n, the total number of processes in the distributed computation, for this property to hold [2].

Event counting

If d is always 1 in rule **R1**, then the *ith* 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, implementations of causal ordering communication and causal distributed shared memory, establishment of global breakpoints, and in determining the consistency of checkpoints in optimistic recovery.

A brief historical perspective of vector clocks

Although the theory associated with vector clocks was first developed in 1988 independently by Fidge and Mattern, vector clocks were informally introduced and used by several researchers before this. Parker *et al.* [17] used a rudimentary vector clocks system to detect inconsistencies of replicated files due to network partitioning. Liskov and Ladin [11] proposed a vector clock system to define highly available distributed services. Similar system of clocks was used by Strom and Yemini [26] to keep track of the causal dependencies between events in their optimistic recovery algorithm and by Raynal to prevent drift between logical clocks [18]. Singhal [24] used vector clocks coupled with a boolean vector to determine the currency of a critical section execution request by detecting the cusality relation between a critical section request and its execution.

3.4.3 On the size of vector clocks

An important question to ask is whether vector clocks of size n are necessary in a computation consisting of n processes. To answer this, we examine the usage of vector clocks.

- A vector clock provides the latest known local time at each other process. If this information in the clock is to be used to explicitly track the progress at every other process, then a vector clock of size n is necessary.

- A popular use of vector clocks is to determine the causality between a pair of events. Given any events e and f, the test for $e \prec f$ if and only if $T(e) < T(f)$, which requires a comparison of the vector clocks of e and f. Although it appears that the clock of size n is necessary, that is not quite accurate. It can be shown that a size equal to the dimension of the partial order $(E, \prec)$ is necessary, where the upper bound on this dimension is n. This is explained below.

To understand this result on the size of clocks for determining causality between a pair of events, we first introduce some definitions. A *linear extension* of a partial order $(E, \prec)$ is a linear ordering of E that is consistent with the partial order, i.e., if two events are ordered in the partial order, they are also ordered in the linear order. A linear extension can be viewed as projecting all the events from the different processes on a single time axis. However, the linear order will necessarily introduce ordering between each pair of events, and some of these orderings are not in the partial order. Also observe that different linear extensions are possible in general. Let $\mathcal{P}$ denote the set of tuples in the partial order defined by the causality relation; so there is a tuple (e, f) in $\mathcal{P}$ for each pair of events e and f such that $e \prec f$. Let $\mathcal{L}_1$, $\mathcal{L}_2$ ... denote the sets of tuples in different linear extensions of this partial order. The set $\mathcal{P}$ is contained in the set obtained by taking the intersection of any such collection of linear extensions $\mathcal{L}_1$, $\mathcal{L}_2$ .... This is because each $\mathcal{L}_i$ must contain all the tuples, i.e., causality dependencies, that are in $\mathcal{P}$. The *dimension* of a partial order is the minimum number of linear extensions whose intersection gives exactly the partial order.

Consider a client–server interaction between a pair of processes. Queries to the server and responses to the client occur in strict alternating sequences. Although $n = 2$, all the events are strictly ordered, and there is only one linear order of all the events that is consistent with the "partial" order. Hence the dimension of this "partial order" is 1. A scalar clock such as one implemented by Lamport's scalar clock rules is adequate to determine $e \prec f$ for any events e and f in this execution.

Now consider an execution on processes P_1 and P_2 such that each sends a message to the other before receiving the other's message. The two send events are concurrent, as are the two receive events. To determine the causality between the send events or between the receive events, it is not sufficient to use a single integer; a vector clock of size $n = 2$ is necessary. This execution exhibits the graphical property called a *crown*, wherein there are some messages $m_0, \ldots m_{n-1}$ such that $Send(m_i) \prec Receive(m_{i+1 \bmod (n-1)})$ for all i from 0 to $n-1$. A crown of n messages has dimension n. We introduced the notion of crown and studied its properties in Chapter 6.

For a complex execution, it is not straightforward to determine the dimension of the partial order. Figure 3.3 shows an execution involving four processes. However, the dimension of this partial order is two. To see this

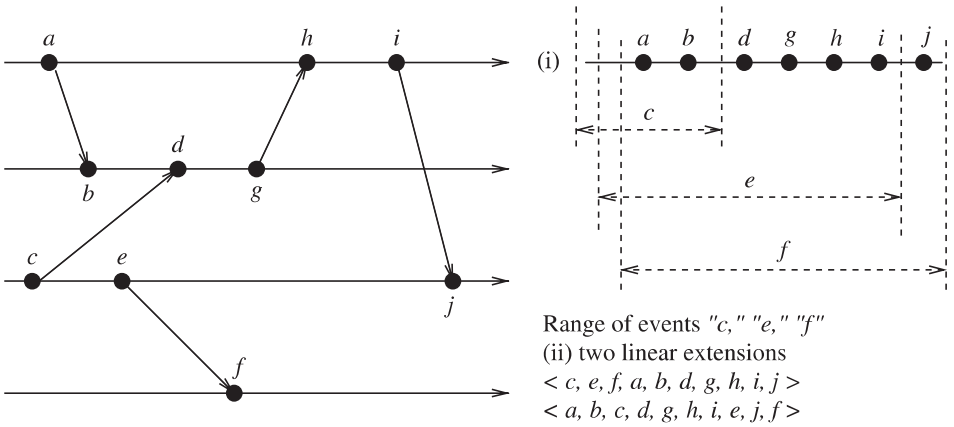

Figure 3.3 Example illustrating dimension of a execution $(E, \prec)$. For $n = 4$ processes, the dimension is 2.

informally, consider the longest chain $\langle a, b, d, g, h, i, j \rangle$. There are events outside this chain that can yield multiple linear extensions. Hence, the dimension is more than 1. The right side of Figure 3.3 shows the earliest possible and the latest possible occurrences of the events not in this chain, with respect to the events in this chain. Let $\mathcal{L}_1$ be $\langle c, e, f, a, b, d, g, h, i, j \rangle$, which contains the following tuples that are not in $\mathcal{P}$:

$$(c, a), (c, b), (c, d), (c, g), (c, h), (c, i), (c, j),$$
$$(e, a), (e, b), (e, d), (e, g), (e, h), (e, i), (e, j),$$
$$(f, a), (f, b), (f, d), (f, g), (f, h), (f, i), (f, j).$$

Let $\mathcal{L}_2$ be $\langle a, b, c, d, g, h, i, e, j, f \rangle$, which contains the following tuples not in $\mathcal{P}$:

$$(a, c), (b, c), (c, d), (c, g), (c, h), (c, i), (c, j),$$
$$(a, e), (b, e), (d, e), (g, e), (h, e), (i, e), (e, j),$$
$$(a, f), (b, f), (d, f), (g, f), (h, f), (i, f), (j, f).$$

Further, observe that $(\mathcal{L}_1 \setminus P) \bigcap \mathcal{L}_2 = \emptyset$ and $(\mathcal{L}_2 \setminus P) \bigcap \mathcal{L}_1 = \emptyset$. Hence, $\mathcal{L}_1 \bigcap \mathcal{L}_2 = \mathcal{P}$ and the dimension of the execution is 2 as these two linear extensions are enough to generate $\mathcal{P}$.

Unfortunately, it is not computationally easy to determine the dimension of a partial order. To exacerbate the problem, the above form of analysis has to be completed a posteriori (i.e., off-line), once the entire partial order has been determined after the completion of the execution.

3.5 Efficient implementations of vector clocks

If the number of processes in a distributed computation is large, then vector clocks will require piggybacking of huge amount of information in messages for the purpose of disseminating time progress and updating clocks. The

message overhead grows linearly with the number of processors in the system and when there are thousands of processors in the system, the message size becomes huge even if there are only a few events occurring in few processors. In this section, we discuss efficient ways to maintain vector clocks; similar techniques can be used to efficiently implement matrix clocks.

Charron-Bost showed [2] that if vector clocks have to satisfy the strong consistency property, then in general vector timestamps must be at least of size n, the total number of processes. Therefore, in general the size of a 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 clocks efficiently [19].

3.5.1 Singhal–Kshemkalyani's differential technique

Singhal–Kshemkalyani's differential technique [25] is based on the observation that between successive message sends to the same process, only a few entries of the vector clock at the sender process are likely to change. This is more likely when the number of processes is large because only a few of them will interact frequently by passing messages. In this 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 sent to p_j.

The technique works as follows: if entries $i_1, i_2, \ldots, i_{n_1}$ of the vector clock at p_i have changed to $v_1, v_2, \ldots, v_{n_1}$, respectively, since the last message sent to p_j, then process p_i piggybacks a compressed timestamp of the form

$$\{(i_1, v_1), (i_2, v_2), \ldots, (i_{n_1}, v_{n_1})\}$$

to the next message to p_j. When p_j receives this message, it updates its vector clock as follows:

$$vt_i[i_k] = max(vt_i[i_k], v_k) \text{ for } k = 1, 2, \ldots, n_1.$$

Thus this technique cuts down the message size, communication bandwidth and buffer (to store messages) requirements. In the worst of case, every element of the vector clock has been updated at p_i since the last message to process p_j, and the next message from p_i to p_j will need to carry the entire vector timestamp of size n. However, on the average the size of the timestamp on a message will be less than n. Note that implementation of this technique requires each process to remember the vector timestamp in the message last sent to every other process. Direct implementation of this will result in $O(n^2)$ storage overhead at each process. This technique also requires that the communication channels follow FIFO discipline for message delivery.

Singhal and Kshemkalyani developed a clever technique that cuts down this storage overhead at each process to $O(n)$. The technique works in

the following manner: process p_i maintains the following two additional vectors:

- $LS_i[1\ldots n]$ ('Last Sent'):
  $LS_i[j]$ indicates the value of $vt_i[i]$ when process p_i last sent a message to process p_j.
- $LU_i[1\ldots n]$ ('Last Update'):
  $LU_i[j]$ indicates the value of $vt_i[i]$ when process p_i last updated the entry $vt_i[j]$.

Clearly, $LU_i[i] = vt_i[i]$ at all times and $LU_i[j]$ needs to be updated only when the receipt of a message causes p_i to update entry $vt_i[j]$. Also, $LS_i[j]$ needs to be updated only when p_i sends a message to p_j. Since the last communication from p_i to p_j, only those elements k of vector clock $vt_i[k]$ have changed for which $LS_i[j] < LU_i[k]$ holds. Hence, only these elements need to be sent in a message from p_i to p_j. When p_i sends a message to p_j, it sends only a set of tuples,

$$\{(x, vt_i[x]) | LS_i[j] < LU_i[x]\},$$

as the vector timestamp to p_j, instead of sending a vector of n entries in a message.

Thus the entire vector of size n is not sent along with a message. Instead, only the elements in the vector clock that have changed since the last message send to that process are sent in the format $\{(p_1, latest\_value), (p_2, latest\_value), \ldots\}$, where p_i indicates that the p_ith component of the vector clock has changed.

This method is illustrated in Figure 3.4. For instance, the second message from p_3 to p_2 (which contains a timestamp $\{(3, 2)\}$) informs p_2 that the third component of the vector clock has been modified and the new value is 2. This is because the process p_3 (indicated by the third component of

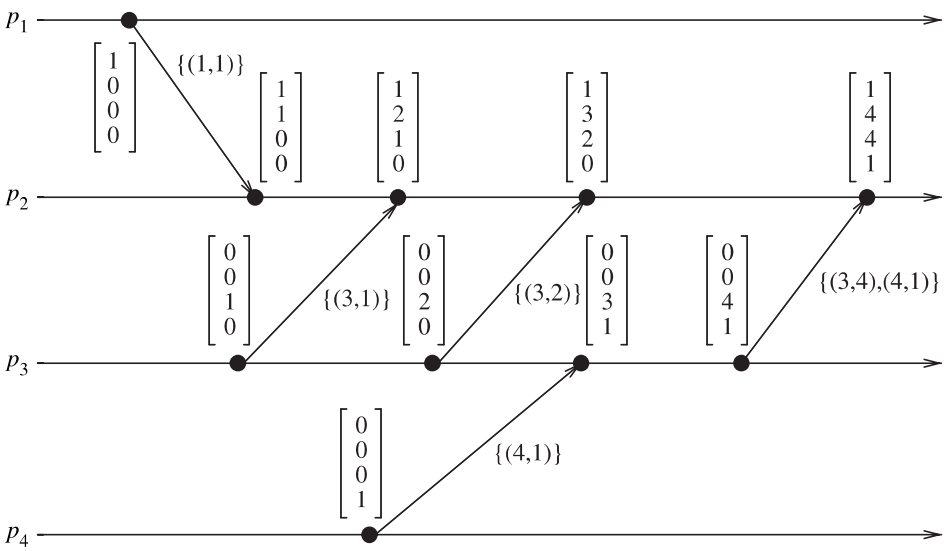

Figure 3.4 Vector clocks progress in Singhal–Kshemkalyani technique [19].

the vector) has advanced its clock value from 1 to 2 since the last message sent to p_2.

The cost of maintaining vector clocks in large systems can be substantially reduced by this technique, especially if the process interactions exhibit temporal or spatial localities. This technique would turn advantageous in a variety of applications including causal distributed shared memories, distributed deadlock detection, enforcement of mutual exclusion and localized communications typically observed in distributed systems.

3.5.2 Fowler–Zwaenepoel's direct-dependency technique

Fowler–Zwaenepoel direct dependency technique [6] reduces the size of messages by transmitting only a scalar value in the messages. No vector clocks are maintained on-the-fly. Instead, a process only maintains information regarding direct dependencies on other processes. A vector time for an event, which represents transitive dependencies on other processes, is constructed off-line from a recursive search of the direct dependency information at processes.

Each process p_i maintains a dependency vector D_i. Initially,

$$D_i[j] = 0 \text{ for } j = 1, \ldots, n.$$

D_i is updated as follows:

1. Whenever an event occurs at p_i, $D_i[i] := D_i[i] + 1$. That is, the vector component corresponding to its own local time is incremented by one.
2. When a process p_i sends a message to process p_j, it piggybacks the updated value of $D_i[i]$ in the message.
3. 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 D_i reflects only 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 3.5 illustrates the Fowler–Zwaenepoel technique. For instance, when process p_4 sends a message to process p_3, it piggybacks a scalar that indicates the direct dependency of p_3 on p_4 because of this message. Subsequently, process p_3 sends a message to process p_2 piggybacking a scalar to indicate the direct dependency of p_2 on p_3 because of this message. Now, process p_2 is in fact indirectly dependent on process p_4 since process p_3 is dependent on process p_4. However, process p_2 is never informed about its indirect dependency on p_4.

Thus although the direct dependencies are duly informed to the receiving processes, the transitive (indirect) dependencies are not maintained by

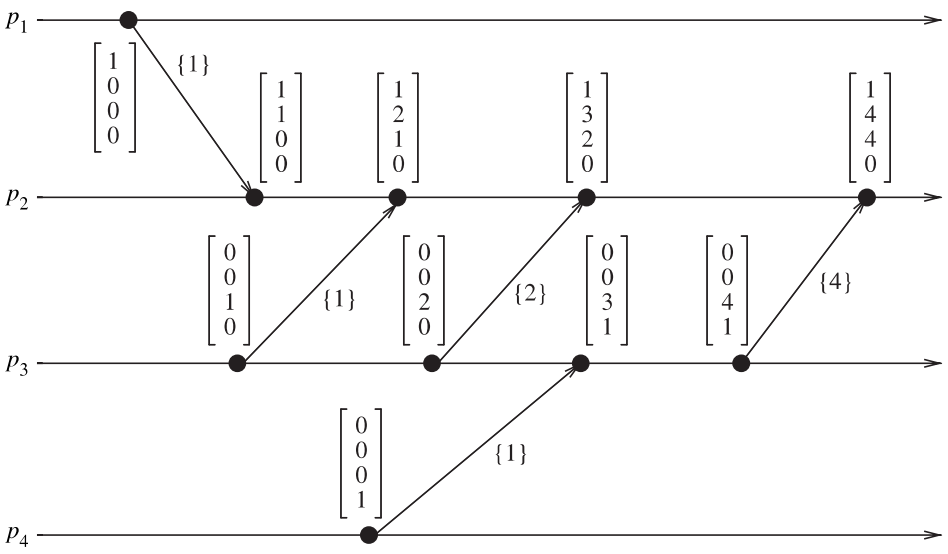

Figure 3.5 Vector clock progress in Fowler–Zwaenepoel technique [19].

this method. They can be obtained only by recursively tracing the direct-dependency vectors of the events off-line. This involves computational overhead and latencies. Thus this method is ideal only for those applications that do not require computation of transitive dependencies on the fly. The computational overheads characteristic of this method makes it best suitable for applications like causal breakpoints and asynchronous checkpoint recovery where computation of causal dependencies is performed offline.

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 timestamp). 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).

The transitive dependencies could be determined by combining an event's direct dependency with that of its directly dependent event. In Figure 3.5, the fourth event of process p_3 is dependent on the first event of process p_4 and the fourth event of process p_2 is dependent on the fourth event of process p_3. By combining these two direct dependencies, it is possible to deduce that the fourth event of process p_2 depends on the first event of process p_4. It is important to note that if event e_j at process p_j occurs before event e_i at process p_i, then all the events from e_0 to e_{j-1} in process p_j also happen before e_i. Hence, it is sufficient to record for e_i the latest event of process p_j that happened before e_i. This way, each event would record its dependencies on

the latest event on every other process it depends on and those events maintain their own dependencies. Combining all these dependencies, the entire set of events that a particular event depends on could be determined off-line.

The off-line computation of transitive dependencies can be performed using a recursive algorithm proposed in [6] and is illustrated in a modified form in Algorithm 3.1. *DTV* is the dependency-tracking vector of size n (where n is the number of process) which is supposed to track all the causal dependencies of a particular event e_i in process p_i. The algorithm then needs to be invoked as *DependencyTrack*$(i, D_i^e[i])$. The algorithm initializes *DTV* to the least possible timestamp value which is 0 for all entries except i for which the value is set to $D_i^e[i]$:

$$\text{for all } k = 1, \ldots, n \text{ and } k \neq i,\ DTV[k]{=}0 \text{ and } DTV[i]{=}D_i^e[i].$$

The algorithm then calls the *VisitEvent* algorithm on process p_i and event e_i. *VisitEvent* checks all the entries $(1, \ldots, n)$ of *DTV* and D_i^e and if the value in D_i^e is greater than the value in *DTV* for that entry, then *DTV* assumes the value of D_i^e for that entry. This ensures that the latest event in process j that e_i depends on is recorded in *DTV*. *VisitEvent* is recursively called on all entries that are newly included in *DTV* so that the latest dependency information can be accurately tracked.

Let us illustrate the recursive dependency trace algorithm by by tracking the dependencies of fourth event at process p_2. The algorithm is invoked as

---

DependencyTrack$(i : process,\ \sigma : event\ index)$
* Casual distributed breakpoint for σ_i *\
* *DTV* holds the result *\
for all $k \neq i$ **do**
  $DTV[k]{=}0$
end for
$DTV[i]{=}\sigma$
end DependencyTrack

VisitEvent$(j : process,\ e : event\ index)$
* Place dependencies of τ into *DTV* *\
for all $k \neq j$ **do**
  $\alpha = D_j^e[k]$
  if $\alpha > DTV[k]$ **then**
    $DTV[k]{=}\alpha$
    VisitEvent(k, α)
  end if
end for
end VisitEvent

Algorithm 3.1 Recursive dependency trace algorithm

DependencyTrack$(2, 4)$. *DTV* is initially set to $< 0\ 4\ 0\ 0 >$ by *DependencyTrack*. It then calls *VisitEvent*$(2, 4)$. The values held by D_2^4 are $< 1\ 4\ 4\ 0 >$. So, *DTV* is now updated to $< 1\ 4\ 0\ 0 >$ and *VisitEvent*$(1, 1)$ is called. The values held by D_1^1 are $< 1\ 0\ 0\ 0 >$. Since none of the entries are greater than those in *DTV*, the algorithm returns. Again the values held by D_2^4 are checked and this time entry 3 is found to be greater in D_2^4 than *DTV*. So, *DTV* is updated as $< 1\ 4\ 4\ 0 >$ and *VisiEvent*$(3, 4)$ is called. The values held by D_3^4 are $< 0\ 0\ 4\ 1 >$. Since entry 4 of D_3^4 is greater than that of *DTV*, it is updated as $< 1\ 4\ 4\ 1 >$ and *VisitEvent*$(4, 1)$ is called. Since none of the entries in D_4^1: $< 1\ 0\ 0\ 0 >$ are greater than those of *DTV*, the algorithm returns to *VisitEvent*$(2, 4)$. Since all the entries have been checked, *VisitEvent*$(2, 4)$ is exited and so is *DependencyTrack*. At this point, *DTV* holds $< 1\ 4\ 4\ 1 >$, meaning event 4 of process p_2 is dependent upon event 1 of process p_1, event 4 of process p_3 and event 1 in process p_4. Also, it is dependent on events that precede event 4 of process p_3 and these dependencies could be obtained by invoking the *DependencyTrack* algorithm on fourth event of process p_3. Thus, all the causal dependencies could be tracked off-line.

This technique can result in a considerable saving of cost since only one scalar is piggybacked on every message. One of the important requirements is that a process updates and records its dependency vectors after receiving a message and before sending out any message. Also, if events occur frequently, this technique will require recording the history of a large number of events.

3.6 Jard–Jourdan's adaptive technique

The Fowler–Zwaenepoel direct-dependency technique does not allow the transitive dependencies to be captured in real time during the execution of processes. In addition, 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 [8], events can be adaptively observed while maintaining the capability of retrieving all the causal dependencies of an observed event. (Observing an event means recording of the information about its dependencies.) This method uses the idea that when an observed event e records its dependencies, then events that follow can determine their transitive dependencies, that is, the set of events that they indirectly depend on, by making use of the information recorded about e. The reason is that when an event e is observed, the information about the send and receive of messages maintained by a process is recorded in that event and the information maintained by the process is then reset and updated. So, when the process

propagates information after e, it propagates only history of activities that took place after e. The next observed event either in the same process or in a different one, would then have to look at the information recorded for e to know about the activities that happened before e. This method still does not allow determining all the causal dependencies in real time, but avoids the problem of recording a large amount of history which is realized when using the direct dependency technique.

To implement the technique of recording the information in an observed event and resetting the information managed by a process, Jard–Jourdan defined a *pseudo-direct* relation $\ll$ on the events of a distributed computation as follows:

If events e_i and e_j happen at process 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 the process p_j and ends before e_i on the process e_i such that there is no observed event on the path. The relation is termed pseudo-direct because event e_i may depend upon many unobserved events on the path, say ue_1, $ue_2, \ldots, ue_n$, etc., which are in turn dependent on each other. If e_i happens after ue_n, then e_i is still considered to be directly dependent upon ue_1, $ue_2, \ldots, ue_n$, since these events are unobserved, which is a falsely assumed to have direct dependency. If another event e_k happens after e_i, then the transitive dependencies of e_k on ue_1, $ue_2, \ldots, ue_n$ can be determined by using the information recorded at e_i and e_i can do the same with e_j.

The technique is implemented using the following mechanism: 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=$\{(i, 0)\}$.

Let $p\_vt_i = \{(i_1, v_1), \ldots, (i, v), \ldots (i_n, v_n\}$ denote the current partial vector clock at process p_i. Let e_vt_i be a variable that holds the timestamp of the observed event.

(i) Whenever an event is observed at process p_i, the contents of the partial vector clock p_vt_i are transferred to e_vt_i and p_vt_i is reset and updated as follows:

$$e\_vt_i = \{(i_1, v_1), \ldots, (i, v), \ldots, (i_n, v_n)\}$$
$$p\_vt_i = \{(i, v+1)\}.$$

(ii) When process p_j sends a message to p_i, it piggybacks the current value of p_vt_j in the message.

(iii) 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=$\{(i_{m1}, v_{m1}),\ldots,(i_{mk}, v_{mk})\}$ 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;
- all $(i_x, max(v_x, v_{mx}))$ for all $(v_x, .)$ that appear in v_pt and v_pt_i.

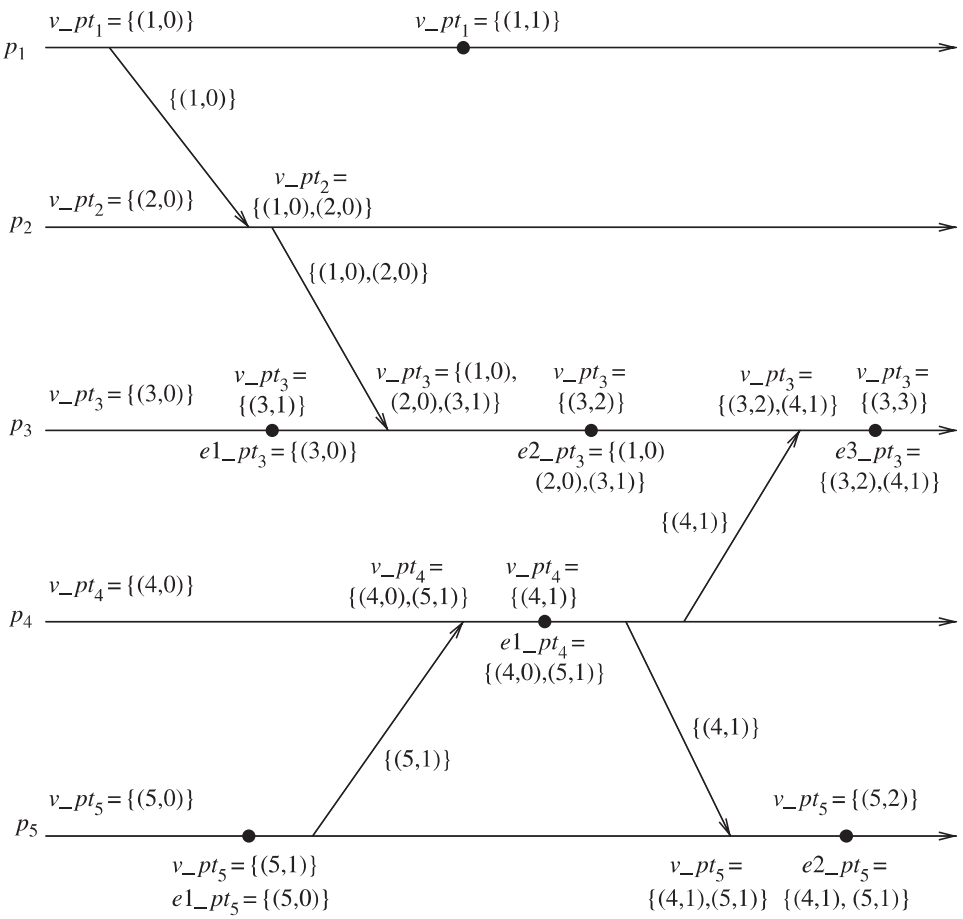

Figure 3.6 Vector clocks progress in the Jard–Jourdan technique [19].

In Figure 3.6, eX_pt_n denotes the timestamp of the Xth observed event at process p_n. For instance, the event 1 observed at p_4 is timestamped $e1\_pt_4 = \{(4, 0), (5, 1)\}$; this timestamp means that the pseudo-direct predecessors of this event are located at process p_4 and p_5, and are respectively the event 0 observed at p_4 and event 1 observed at p_5. v_pt_n denotes a list of timestamps collected by a process p_n for the unobserved events and is reset and updated after an event is observed at p_n. For instance, let us consider v_pt_3. Process p_3 first collects the timestamp of event zero $(3, 0)$ into v_pt_3 and when the observed event 1 occurs, it transfers its content to $e1\_pt_3$, resets its list and updates its value to $(3, 1)$ which is the timestamp of the observed event. When it receives a message from process p_2, it includes those elements that are not already present in its list, namely, $(1, 0)$ and $(2, 0)$ to v_pt_3. Again, when event 2 is observed, it resets its list to $\{(3, 2)\}$ and transfers its content to $e2\_pt_3$ which holds $\{(1, 0), (2, 0), (3, 1)\}$. It can be seen that event 2 at process p_3 is directly dependent upon event 0 on process p_2 and event 1 on process p_3. But, it is pseudo-directly dependent upon event 0 at process p_1. It also depends on event 0 at process p_3 but this dependency information is obtained by examining $e1\_pt_3$ recorded by the observed event. Thus, transitive dependencies of event 2 at process p_3 can be computed by examining the observed events in $e2\_pt_3$. If this is done recursively, then all the causal

dependencies of an observed event can be retrieved. It is also pertinent to observe here that these transitive dependencies cannot be determined online but from a log of the events.

This method can help ensure that the list piggybacked on a message is of optimal size. It is also possible to limit the size of the list by introducing a dummy observed event. If the size of the list is to be limited to k, then when timestamps of k events have been collected in the list, a dummy observed event can be introduced to receive the contents of the list. This allows a lot of flexibility in managing the size of messages.

3.7 Matrix time

3.7.1 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;
- $mt_i[i, j]$ denotes the latest knowledge that process p_i has about the local logical clock, $mt_j[j, j]$, of process p_j (note that row, $mt_i[i, .]$ is nothing but the vector clock $vt_i[.]$ and exhibits all the properties of vector clocks);
- $mt_i[j, k]$ represents the knowledge that process p_i has 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 global logical 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:

- **R1**: Before executing an event, process p_i 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:
  - (i) update its global logical time as follows:
    - (a) $1 \leq k \leq n : mt_i[i, k] := max(mt_i[i, k], mt[j, k])$, (that is, update its row $mt_i[i, *]$ with p_j's row in the received timestamp, mt);
    - (b) $1 \leq k, l \leq n : mt_i[k, l] := max(mt_i[k, l], mt[k, l])$;
  - (ii) execute **R1**;
  - (iii) deliver message m.

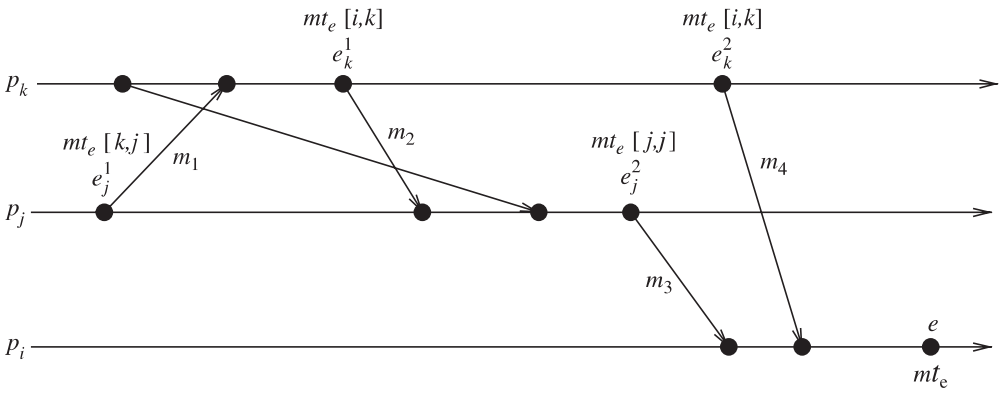

Figure 3.7 Evolution of matrix time [19].

Figure 3.7 gives an example to illustrate how matrix clocks progress in a distributed computation. We assume $d = 1$. 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 events at process p_k, and e_j^1 and e_j^2 which are the x_j^1th and x_j^2th events at p_j. Let mt_e denote the matrix timestamp associated with event e. Due to message m_4, e_k^2 is the last event of p_k that causally precedes e, therefore, we have $mt_e[i, k] = mt_e[k, k] = x_k^2$. Likewise, $mt_e[i, j] = mt_e[j, j] = x_j^2$. The last event of p_k known by p_j, to the knowledge of p_i when it executed event e, is e_k^1; therefore, $mt_e[j, k] = x_k^1$. Likewise, we have $mt_e[k, j] = x_j^1$.

A system of matrix clocks was first informally proposed by Michael and Fischer [5] and has been used by Wuu and Bernstein [28] and by Sarin and Lynch [22] to discard obsolete information in replicated databases.

3.7.2 Basic properties

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

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

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.

3.8 Virtual time

The virtual time system is a paradigm for organizing and synchronizing distributed systems using virtual time [7]. This section provides a description

of virtual time and its implementation using the time warp mechanism (a lookahead-rollback synchronization mechanism using rollback via antimessages).

The implementation of virtual time using the time warp mechanism works on the basis of an optimistic assumption. Time warp relies on the general lookahead-rollback mechanism where each process executes without regard to other processes having synchronization conflicts. If a conflict is discovered, the offending processes are rolled back to the time just before the conflict and executed forward along the revised path. Detection of conflicts and rollbacks are transparent to users. The implementation of virtual time using the time warp mechanism makes the following optimistic assumption: synchronization conflicts and thus rollback generally occurs rarely.

In the following sections, we discuss in detail virtual time and how the time warp mechanism is used to implement it.

3.8.1 Virtual time definition

Virtual time is a global, one-dimensional, temporal coordinate system on a distributed computation to measure the computational progress and to define synchronization. A virtual time system is a distributed system executing in coordination with an imaginary virtual clock that uses virtual time [7]. Virtual times are real values that are totally ordered by the less than relation, "<". Virtual time is implemented as a collection of several loosely synchronized local virtual clocks. As a rule, these local virtual clocks move forward to higher virtual times; however, occasionally they move backwards.

In a distributed system, processes run concurrently and communicate with each other by exchanging messages. Every message is characterized by four values:

(i) *name of the sender;*
(ii) *virtual send time;*
(iii) *name of the receiver;*
(iv) *virtual receive time.*

Virtual send time is the virtual time at the sender when the message is sent, whereas virtual receive time specifies the virtual time when the message must be received (and processed) by the receiver. Clearly, a big problem arises when a message arrives at process late, that is, the virtual receive time of the message is less than the local virtual time at the receiver process when the message arrives.

Virtual time systems are subject to two semantic rules similar to Lamport's clock conditions:

Rule 1 Virtual send time of each message < virtual receive time of that message.

Rule 2 Virtual time of each event in a process < virtual time of next event in that process.

The above two rules imply that a process sends all messages in increasing order of virtual send time and a process receives (and processes) all messages in the increasing order of virtual receive time. Causality of events is an important concept in distributed systems and is also a major constraint in the implementation of virtual time. It is important to know which event caused another one and the one that causes another should be completely executed before the caused event can be processed.

The constraint in the implementation of virtual time can be stated as follows:

If an event A causes event B, then the execution of A and B must be scheduled in real time so that A is completed before B starts.

If event A has an earlier virtual time than event B, we need execute A before B provided there is no causal chain from A to B. Better performance can be achieved by scheduling A concurrently with B or scheduling A after B. If A and B have exactly the same virtual time coordinate, then there is no restriction on the order of their scheduling. If A and B are distinct events, they will have different virtual space coordinates (since they occur at different processes) and neither will be a cause for the other. Hence to sum it up, events with virtual time < "t" complete before the starting of events at time "t" and events with virtual time > "t" will start only after events at time "t" are complete.

Characteristics of virtual time

1. Virtual time systems are not all isomorphic; they may be either discrete or continuous.
2. Virtual time may be only partially ordered (in this implementation, total order is assumed.)
3. Virtual time may be related to real time or may be independent of it.
4. Virtual time systems may be visible to programmers and manipulated explicitly as values, or hidden and manipulated implicitly according to some system-defined discipline
5. Virtual times associated with events may be explicitly calculated by user programs or they may be assigned by fixed rules.

3.8.2 Comparison with Lamport's logical clocks

Lamport showed that in real-time temporal relationships "*happens before*" and "*happens after*," operationally definable within a distributed system, form only a partial order, not a total order, and concurrent events are incomparable under that partial order. He also showed that it is always possible to extend partial order to total order by defining artificial clocks. An artificial clock is created for each process with unique labels from a totally ordered set in a manner consistent with partial order. He also provided an algorithm on how

to accomplish this task of yielding an assignment of totally ordered clock values. In virtual time, the reverse of the above is done by assuming that every event is labeled with a clock value from a totally ordered virtual time scale satisfying Lamport's clock conditions. Thus the time warp mechanism is an inverse of Lamport's scheme.

In Lamport's scheme, all clocks are conservatively maintained so that they never violate causality. A process advances its clock as soon as it learns of new causal dependency. In virtual time, clocks are optimisticaly advanced and corrective actions are taken whenever a violation is detected.

Lamport's initial idea brought about the concept of virtual time but the model failed to preserve causal independence. It was possible to make an analysis in the real world using timestamps but the same principle could not be implemented completely in the case of asynchronous distributed systems for the lack of a common time base.

The implementation of the virtual time concept using the time warp mechanism is easier to understand and reason about than real time.

3.8.3 Time warp mechanism

In the implementation of virtual time using the time warp mechanism, the virtual receive time of a message is considered as its timestamp. The necessary and sufficient conditions for the correct implementation of virtual time are that each process must handle incoming messages in *timestamp* order. This is highly undesirable and restrictive because process speeds and message delays are likely to be highly variable. So it is natural for some processes to get ahead in virtual time of other processes.

Since we assume that virtual times are real numbers, it is impossible for a process on the basis of local information alone to block and wait for the message with the next timestamp. It is always possible that a message with an earlier timestamp arrives later. So, when a process executes a message, it is very difficult for it determine whether a message with an earlier timestamp will arrive later. This is the central problem in virtual time that is solved by the time warp mechanism.

The advantage of the time warp mechanism is that it doesn't depend on the underlying computer architecture and so portability to different systems is easily achieved. However, message communication is assumed to be reliable, but messages may not be delivered in FIFO order.

The time warp mechanism consists of two major parts: local control mechanism and global control mechanism. The local control mechanism ensures that events are executed and messages are processed in the correct order. The global control mechanism takes care of global issues such as global progress, termination detection, I/O error handling, flow control, etc.

3.8.4 The local control mechanism

There is no global virtual clock variable in this implementation; each process has a *local virtual clock* variable. The local virtual clock of a process doesn't change during an event at that process but it changes only between events. On the processing of next message from the input queue, the process increases its local clock to the timestamp of the message. At any instant, the value of virtual time may differ for each process but the value is transparent to other processes in the system.

When a message is sent, the virtual send time is copied from the sender's virtual clock while the name of the receiver and virtual receive time are assigned based on the application-specific context.

All arriving messages at a process are stored in an input queue in increasing order of timestamp (receive times). Ideally, no messages from the past (called late messages) should arrive at a process. However, processes will receive late messages due to factors such as different computation rates of processes and network delays. The semantics of virtual time demands that incoming messages be received by each process strictly in timestamp order. The only way to accomplish this is as follows: on the reception of a late message, the receiver rolls back to an earlier virtual time, cancelling all intermediate side effects and then executes forward again by executing the late message in the proper sequence. If all the messages in the input queue of a process are processed, the state of the process is said to *terminate* and its clock is set to +inf. However, the process is not destroyed as a late message may arrive resulting it to rollback and execute again. The situation can be described by saying that each process is doing a constant "lookahead," processing future messages from its input queue.

Over a length computation, each process may roll back several times while generally progressing forward with rollback completely transparent to other processes in the system. Programmers can thus write correct software without paying much attention to late-arriving messages.

Rollback in a distributed system is complicated by the fact that the process that wants to rollback might have sent many messages to other processes, which in turn might have sent many messages to other processes, and so on, leading to deep side effects. For rollback, messages must be effectively "*unsent*" and their side effects should be undone. This is achieved efficiently by using antimessages.

Antimessages and the rollback mechanism

Runtime representation of a process is composed of the following:

1. **Process name** Virtual spaces coordinate which is unique in the system.
2. **Local virtual clock** Virtual time coordinate
3. **State** Data space of the process including execution stack, program counter, and its own variables

4. **State queue** Contains saved copies of process's recent states as rollback with the time warp mechanism requires the state of the process being saved. It is not necessary to retain states all the way from the beginning of the virtual time, however, the reason for which will be explained later in the global control mechanism.
5. **Input queue** Contains all recently arrived messages in order of virtual receive time. Processed messages from the input queue are not deleted as they are saved in the output queue with a negative sign (antimessage) to facilitate future rollbacks.
6. **Output queue** Contains negative copies of messages that the process has recently sent in virtual send time order. They are needed in case of a rollback.

For every message, there exists an antimessage that is the same in content but opposite in sign. Whenever a process sends a message, a copy of the message is transmitted to the receiver's input queue and a negative copy (antimessage) is retained in the sender's output queue for use in sender rollback.

Whenever a message and its antimessage appear in the same queue, regardless of the order in which they arrived, they immediately annihilate each other resulting in shortening of the queue by one message.

Generally when a message arrives at the input queue of a process with timestamp greater than the virtual clock time of its destination process, it is simply enqueued by the interrupt routine and the running process continues. But when the destination process' virtual time is greater than the virtual time of the message received, the process must do a rollback.

The first step in the rollback mechanism is to search the "state queue" for the last saved state with a timestamp that is less than the timestamp of the message received and restore it. We make the timestamp of the received message as the value of the local virtual clock and discard from the state queue all states saved after this time. Then the execution resumes forward from this point. Now all the messages that are sent between the current state and earlier state must be "unsent." This is taken care of by executing a simple rule:

To unsend a message, simply transmit its antimessage.

This results in antimessages following the positive ones to the destination. A negative message causes a rollback at its destination if its virtual receive time is less than the receiver's virtual time (just as a positive message does).

Depending on the timing, there are several possibilities at the receiver's end:

1. If the original (positive) message has arrived but not yet been processed, its virtual receive time must be greater than the value in the receiver's virtual clock. The negative message, having the same virtual receive time, will be enqueued and will not cause a rollback. It will, however, cause annihilation with the positive message leaving the receiver with no record of that message.

2. The second possibility is that the original positive message has a virtual receive time that is now in the present or past with respect to the receiver's virtual clock and it may have already been partially or completely processed, causing side effects on the receiver's state. In this case, the negative message will also arrive in the receiver's past and cause the receiver to rollback to a virtual time when the positive message was received. It will also annihilate the positive message, leaving the receiver with no record that the message existed. When the receiver executes again, the execution will assume that these message never existed. Note that, as a result of the rollback, the process may send antimessages to other processes.
3. A negative message can also arrive at the destination before the positive one. In this case, it is enqueued and will be annihilated when the positive message arrives. If it is the negative message's turn to be executed at a processs' input queue, the receiver may take any action like a no-op. Any action taken will eventually be rolled back when the corresponding positive message arrives. An optimization would be to skip the antimessage from the input queue and treat it as a no-op, and when the corresponding positive message arrives, it will annihilate the negative message, and inhibit any rollback.

The antimessage protocol has several advantages: it is extremely robust and works under all possible circumstances; it is free from deadlocks as there is no blocking; it is also free from domino effects. In the worst case, all processes in the system rollback to the same virtual time as the original and then proceed forward again.

3.8.5 Global control mechanism

The global control mechanism resolves the following issues:

- System global progress amidst rollback activity?
- Detection of global termination?
- Errors, I/O handling on rollbacks?
- Running out of memory while saving copies of messages?

How these issues are resolved by the global control mechanism will be discussed later; first we discuss the important concept of global virtual time.

Global virtual time

The concept of global virtual time (GVT) is central to the global control mechanism. Global virtual time [14] is a property of an instantaneous global snapshot of system at real time "*r*" and is defined as follows:

Global virtual time (GVT) at real time *r* is the minimum of:

1. all virtual times in all virtual clocks at time *r*; and
2. the virtual send times of all messages that have been sent but have not yet been processed at time "*r*".

GVT is defined in terms of the *virtual send time* of unprocessed messages, instead of the virtual receive time, because of the flow control (discussed below). If every event completes normally, if messages are delivered reliably, if the scheduler does not indefinitely postpone execution of the farthest behind process, and if there is sufficient memory, then GVT will eventually increase.

It is easily shown by induction that the message (sends, arrivals, and receipts) never decreases GVT even though local virtual time clocks roll back frequently. These properties make it appropriate to consider GVT as a virtual clock for the system as a whole and to use it as the measure of system progress. GVT can thus be viewed as a moving commitment horizon: any event with virtual time less than GVT cannot be rolled back and may be committed safely.

It is generally impossible for one time warp mechanism to know at any real time "r," exactly what GVT is. But GVT can be characterized more operationally by its two properties discussed above. This characterization leads to a fast distributed GVT estimation algorithm that takes $O(d)$ time, where "d" is the delay required for one broadcast to all processors in the system. The algorithm runs concurrently with the main computation and returns a value that is between the true GVT at the moment the algorithm starts and the true GVT at the moment of completion. Thus it gives a slightly out-of-date value for GVT which is the best one can get.

During execution of a virtual time system, time warp must periodically estimate GVT. A higher frequency of GVT estimation produces a faster response time and better space utilization at the expense of processor time and network bandwidth.

Applications of GVT

GVT finds several applications in a virtual time system using the time warp mechanism.

Memory management and flow control

An attractive feature of the time warp mechanism is that it is possible to give simple algorithms for managing memory. The time warp mechanism uses the concept of fossil detection where information older than GVT is destroyed to avoid memory overheads due to old states in state queues, messages stored in output queues, "past" messages in input queues that have already been processed, and "future" messages in input queues that have not yet been received.

There is another kind of memory overhead due to future messages in the input queues that have not yet been received. So, if a receiver's memory is full of input messages, the time warp mechanism may be able to recover space by returning an unreceived message to the process that sent it and then rolling back to cancel out the sending event.

Normal termination detection
The time warp mechanism handles the termination detection problem through GVT. A process terminates whenever it runs out of messages and its local virtual clock is set to +inf. Whenever GVT reaches +inf, all local virtual clock variables must read +inf and no message can be in transit. No process can ever again unterminate by rolling back to a finite virtual time. The time warp mechanism signals termination whenever the GVT calculation returns "+inf" value in the system.

Error handling
Not all errors cause termination. Most of the errors can be avoided by rolling back the local virtual clock to some finite value. The error is only "committed" if it is impossible for the process to roll back to a virtual time on or before the error. The committed error is reported to some policy software or to the user.

Input and output
When a process sends a command to an output device, it is important that the physical output activity not be committed immediately because the sending process may rollback and cancel the output request. An output activity can only be performed when GVT exceeds the virtual receive time of the message containing the command.

Snapshots and crash recovery
An entire snapshot of the system at virtual time "t" can be constructed by a procedure in which each process "*snapshots*" itself as it passes virtual time t in the forward direction and "*unsnapshots*" itself whenever it rolls back over virtual time "t". Whenever GVT exceeds "t," the snapshot is complete and valid.

Example: distributed discrete event simulations Distributed discrete event simulation [1, 16, 21] is the most studied example of virtual time systems; every process represents an object in the simulation and virtual time is identified with simulation time. The fundamental operation in discrete event simulation is for one process to schedule an event for execution by another process at a later simulation time. This is emulated by having the first process send a message to the second process with the virtual receive time of the message equal to the event's scheduled time in the simulation. When an event message is received by a process, there are three possibilities: its timestamp is either before, after, or equal to the local value of simulation time.

If its timestamp is after the local time, an input event combination is formed and the appropriate action is taken. However, if the timestamp of the received event message is less than or equal to the local clock value, the process has

already processed an event combination with time greater than or equal to the incoming event. The process must then rollback to the time of the incoming message which is done by an elaborate checkpointing mechanism that allows earlier states to be restored. Essentially an earlier state is restored, input event combinations are rescheduled, and output events are cancelled by sending antimessages. The process has buffers that save past inputs, past states, and antimessages.

Distributed discrete event simulation is one of the most general applications of the virtual time paradigm because the virtual times of events are completely under the control of the user, and because it makes use of almost all the degrees of freedom allowed in the definition of a virtual time system.

3.9 Physical clock synchronization: NTP

3.9.1 Motivation

In centralized systems, there is no need for clock synchronization because, generally, there is only a single clock. A process gets the time by simply issuing a system call to the kernel. When another process after that tries to get the time, it will get a higher time value. Thus, in such systems, there is a clear ordering of events and there is no ambiguity about the times at which these events occur.

In distributed systems, there is no global clock or common memory. Each processor has its own internal clock and its own notion of time. In practice, these clocks can easily drift apart by several seconds per day, accumulating significant errors over time. Also, because different clocks tick at different rates, they may not remain always synchronized although they might be synchronized when they start. This clearly poses serious problems to applications that depend on a synchronized notion of time. For most applications and algorithms that run in a distributed system, we need to know time in one or more of the following contexts:

- The time of the day at which an event happened on a specific machine in the network.
- The time interval between two events that happened on different machines in the network.
- The relative ordering of events that happened on different machines in the network.

Unless the clocks in each machine have a common notion of time, time-based queries cannot be answered. Some practical examples that stress the need for synchronization are listed below:

- In database systems, the order in which processes perform updates on a database is important to ensure a consistent, correct view of the database.

To ensure the right ordering of events, a common notion of time between co-operating processes becomes imperative.

- Liskov [10] states that clock synchronization improves the performance of distributed algorithms by replacing communication with local computation. When a node p needs to query node q regarding a property, it can deduce the property with some previous information it has about node p and its knowledge of the local time in node q.
- It is quite common that distributed applications and network protocols use timeouts, and their performance depends on how well physically dispersed processors are time-synchronized. Design of such applications is simplified when clocks are synchronized.

Clock synchronization is the process of ensuring that physically distributed processors have a common notion of time. It has a significant effect on many problems like secure systems, fault diagnosis and recovery, scheduled operations, database systems, and real-world clock values. It is quite common that distributed applications and network protocols use timeouts, and their performance depends on how well physically dispersed processors are time-synchronized. Design of such applications is simplified when clocks are synchronized.

Due to different clocks rates, the clocks at various sites may diverge with time, and periodically a clock synchrinization must be performed to correct this clock skew in distributed systems. Clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time). Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed *physical clocks*.

3.9.2 Definitions and terminology

We provide the following definitions [13,14]. C_a and C_b are any two clocks.

1. **Time** The time of a clock in a machine p is given by the function $C_p(t)$, where $C_p(t) = t$ for a perfect clock.
2. **Frequency** Frequency is the rate at which a clock progresses. The frequency at time t of clock C_a is $C'_a(t)$.
3. **Offset** Clock offset is the difference between the time reported by a clock and the *real time*. The offset of the clock C_a is given by $C_a(t) - t$. The offset of clock C_a relative to C_b at time $t \geq 0$ is given by $C_a(t) - C_b(t)$.
4. **Skew** The skew of a clock is the difference in the frequencies of the clock and the perfect clock. The skew of a clock C_a relative to clock C_b at time t is $C'_a(t) - C'_b(t)$.
   If the skew is bounded by ρ, then as per Eq.(3.1), clock values are allowed to diverge at a rate in the range of $1 - \rho$ to $1 + \rho$.

5. **Drift (rate)** The drift of clock C_a is the second derivative of the clock value with respect to time, namely, $C_a''(t)$. The drift of clock C_a relative to clock C_b at time t is $C_a''(t) - C_b''(t)$.

3.9.3 Clock inaccuracies

Physical clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time).

However, due to the clock inaccuracy discussed above, a timer (clock) is said to be working within its specification if

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho, \tag{3.1}$$

where constant ρ is the maximum skew rate specified by the manufacturer. Figure 3.8 illustrates the behavior of fast, slow, and perfect clocks with respect to UTC.

Offset delay estimation method

The *Network Time Protocol (NTP)* [15], which is widely used for clock synchronization on the Internet, uses the the *offset delay estimation* method. The design of NTP involves a hierarchical tree of time servers. The primary server at the root synchronizes with the UTC. The next level contains secondary servers, which act as a backup to the primary server. At the lowest level is the synchronization subnet which has the clients.

Clock offset and delay estimation

In practice, a source node cannot accurately estimate the local time on the target node due to varying message or network delays between the nodes. This protocol employs a very common practice of performing several trials and chooses the trial with the minimum delay. Recall that Cristian's remote

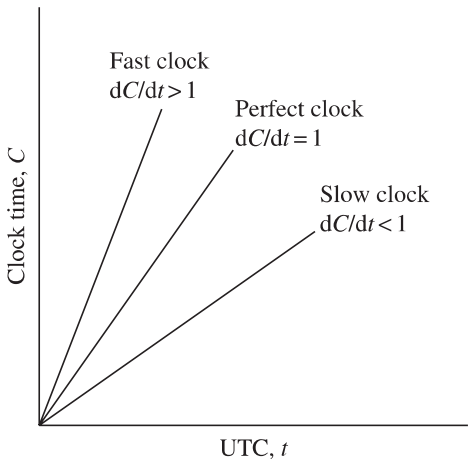

Figure 3.8 The behavior of fast, slow, and perfect clocks with respect to UTC.

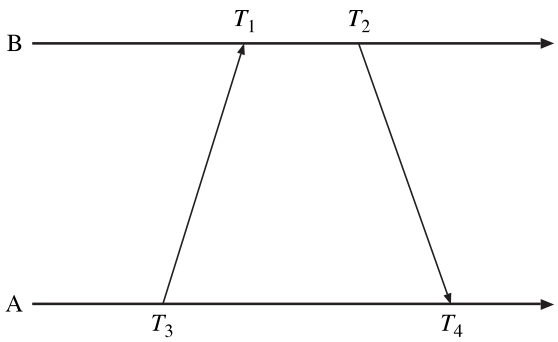

Figure 3.9 Offset and delay estimation [15].

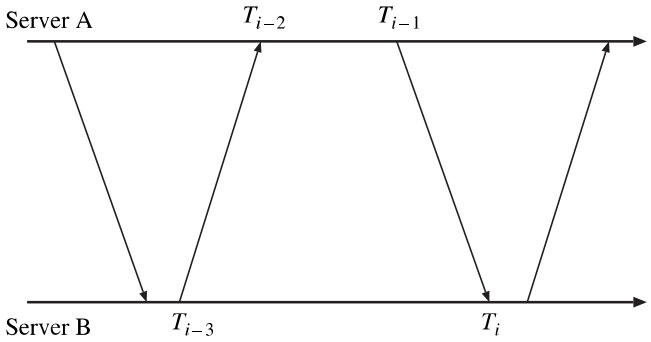

Figure 3.10 Timing diagram for the two servers [15].

clock reading method [3] also relied on the same strategy to estimate message delay.

Figure 3.9 shows how NTP timestamps are numbered and exchanged between peers A and B. Let T_1, T_2, T_3, T_4 be the values of the four most recent timestamps as shown. Assume that clocks A and B are stable and running at the same speed. Let $a = T_1 - T_3$ and $b = T_2 - T_4$. If the network delay difference from A to B and from B to A, called *differential delay*, is small, the clock offset θ and roundtrip delay δ of B relative to A at time T_4 are approximately given by the following:

$$\theta = \frac{a+b}{2}, \quad \delta = a - b. \tag{3.2}$$

Each NTP message includes the latest three timestamps T_1, T_2, and T_3, while T_4 is determined upon arrival. Thus, both peers A and B can independently calculate delay and offset using a single bidirectional message stream as shown in Figure 3.10. The NTP protocol is shown in Figure 3.11.

3.10 Chapter summary

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

- A pair of servers in symmetric mode exchange pairs of timing messages.
- A store of data is then built up about the relationship between the two servers (pairs of offset and delay).

  Specifically, assume that each peer maintains pairs (O_i,D_i), where:

  O_i – measure of offset (θ)
  D_i – transmission delay of two messages (δ).

- The offset corresponding to the minimum delay is chosen. Specifically, the delay and offset are calculated as follows. Assume that message m takes time t to transfer and m' takes t' to transfer.
- The offset between A's clock and B's clock is O. If A's local clock time is $A(t)$ and B's local clock time is $B(t)$, we have

$$A(t) = B(t) + O. \tag{3.3}$$

Then,

$$T_{i-2} = T_{i-3} + t + O, \tag{3.4}$$

$$T_i = T_{i-1} - O + t'. \tag{3.5}$$

Assuming $t = t'$, the offset O_i can be estimated as

$$O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2. \tag{3.6}$$

The round-trip delay is estimated as

$$D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}). \tag{3.7}$$

- The eight most recent pairs of (O_i, D_i) are retained.
- The value of O_i that corresponds to minimum D_i is chosen to estimate O.

Figure 3.11 The network time protocol (NTP) synchronization protocol [15].

systems and it is possible only to realize an approximation of it. Typically, 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. Causality among events in a distributed system is a powerful concept in reasoning, analyzing, and drawing inferences about a computation.

We presented a general framework of logical clocks in distributed systems and discussed three systems of logical clocks, namely, scalar, vector, and matrix clocks, that have been proposed to capture causality between events of

a distributed computation. These systems of clocks have been used to solve a variety of problems in distributed systems such as distributed algorithms 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 the compuatation 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 the compuatation overheads are likely to be high; however, vector clocks possess a powerful property – there is an isomorphism between the set of partially ordered events in a distributed computation and their vector timestamps. This is a very useful and interesting property of vector clocks that 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 the compuatation overheads are high; however, matrix clocks are very 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 very useful in aplications 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 the overheads.

We discussed three efficient implementations of vector clocks; similar techniques can be used to efficiently implement matrix clocks. Singhal–Kshemkalyani's differential technique exploits the fact that, between successive events at a process, only few entries of its vector clock are likely to change. Thus, when a process p_i sends a message to a process p_j, it piggybacks only those entries of its vector clock that have changed since the last message send to p_j, reducing the communication and buffer (to store messages) overheads. Fowler–Zwaenepoel's direct-dependency technique does not maintain vector clocks on-the-fly. Instead, a process only maintains information regarding direct dependencies on other processes. A vector timestamp 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. Thus, the technique has low run-time overhead. In the Fowler–Zwaenepoel technique, however, a process must update and record its dependency vector after receiving a message but before sending out any message. If events occur very frequently, this technique will require recording the history of a large number of events. In the Jard–Jourdan technique, events can be adaptively observed while maintaining the capability of retrieving all the causal dependencies of an observed event.

Virtual time system is a paradigm for organizing and synchronizing distributed systems using virtual time. We discussed virtual time and its implementation using the time warp mechanism.

3.11 Exercises

Exercise 3.1 Why is it difficult to keep a synchronized system of physical clocks in distributed systems?

Exercise 3.2 If events corresponding to vector timestamps Vt_1, Vt_2, ..., Vt_n are mutually concurrent, then prove that

$$(Vt_1[1], Vt_2[2], \ldots . Vt_n[n]) = max(Vt_1, Vt_2, \ldots ., Vt_n) .$$

Exercise 3.3 If events e_i and e_j respectively occurred at processes p_i and p_j and are assigned vector timestamps VT_{e_i} and VT_{e_j}, respectively, then show that

$$e_i \rightarrow e_j \Leftrightarrow VT_{e_i}[i] < VT_{e_j}[i].$$

Exercise 3.4 The size of matrix clocks is quadratic with respect to the system size. Hence the message overhead is likely to be substantial. Propose a technique for matrix clocks similar to that of Singhal–Kshemkalyani to decrease the volume of information transmitted in messages and stored at processes.

3.12 Notes on references

The idea of logical time was proposed by Lamport in 1978 [9] in an attempt to order events in distributed systems. He also suggested an implementation of logical time as a scalar time. Vector clocks were developed independently by Fidge [4], Mattern [12], and Schmuck [23]. Charron-Bost formally showed [2] that if vector clocks have to satisfy the strong consistency property, then the length of vector timestamps must be at least n. Efficient implementations of vector clocks can be found in [8,25]. Matrix clocks was informally proposed by Michael and Fischer [7] and used by Wuu and Bernstein [28] and by Lynch and Sarin [22] to discard obsolete information. Raynal and Singhal present a survey of scalar, vector, and matrix clocks in [19]. More details on virtual time can be found in a classical paper by Jefferson [7]. A survey of physical clock synchronization in wireless sensor networks can be found in [27].

References

[1] B. R. Preiss, The Yaddes distributed discrete event simulation specification language and execution environments, *Proceedings of the SCS Multiconference on Distributed Simulation*, 1989, 139–144.
[2] B. Charron-Bost, Concerning the size of logical clocks in distributed systems, *Information Processing Letters*, **39**, 1991, 11–16.
[3] F. Cristian, Probabilistic clock synchronization, *Distributed Computing,* **3**, 1989, 146–158.
[4] C. Fidge, Logical time in distributed computing systems, *IEEE Computer*, August, 1991, 28–33.

[5] M. J. Fischer and A. Michael, Sacrifying serializability to attain hight availability of data in an unreliable network, *Proceedings of the ACM Symposium on Principles of Database Systems*, 1982, 70–75.

[6] J. Fowler and W. Zwaenepoel, Causal distributed breakpoints, *Proceedings of the 10th International Conference on Distributed Computing Systems*, 1990, 134–141.

[7] D. Jefferson, Virtual time, *ACM Toplas*, **7**(3), 1985, 404–425.

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

[9] L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM*, **21**, 1978, 558–564.

[10] B. Liskov, Practical uses of synchronized clocks in distributed systems, *Proceedings of Tenth Annual ACM Symposium on Principles of Distributed Computing*, August 1991, pp. 1–9.

[11] B. Liskov and R. Ladin, Highly available distributed services and fault-tolerant distributed garbage collection, *Proceedings of the 5th ACM Symposium on PODC*, 1986, 29–39.

[12] F. Mattern, Virtual time and global states of distributed systems, in Cosnard, Q and Raynal, R. (eds) *Proceedings of the Parallel and Distributed Algorithms Conference*, North-Holland, 1988, 215–226.

[13] D. L. Mills, *Network Time Protocol (version 3): Specification, Implementation, and Analysis*, Technical Report, Network Information Center, SRI International, Menlo Park, CA, March, 1992.

[14] D. L. Mills, *Modelling and Analysis of Computer Network Clocks*, Technical Report, 92-5-2, Electrical Engineering Department, University of Delaware, May, 1992.

[15] D. L. Mills, Internet time synchronization: the network time protocol, *IEEE Transactions on Communications*, **39**(10), 1991, 1482–1493.

[16] J. Misra, Distributed discrete event simulation, *ACM Computing Surveys*, **18**(1), 1986, 39–65.

[17] D. S. Parker *et al.*, Detection of mutual inconsistency in distributed systems, *IEEE Transactions on Software Engineeing,* **9**(3), 1983, 240–246.

[18] M. Raynal, A distributed algorithm to prevent mutual drift between n logical clocks, *Information Processing Letters*, **24**, 1987, 199–202.

[19] M. Raynal and M. Singhal, Logical time: capturing causality in distributed systems, *IEEE Computer*, **30**(2), 1996, 49–56.

[20] G. Ricart, and A. K. Agrawala, An optimal algorithm for mutual exclusion in computer networks, *Communications of the ACM*, **24**(1), 1981, 9–17

[21] R. Righter and J. C. Walrand, Distributed simulation of discrete event systems, *Proceedings of the IEEE*, 1988, and 99–113.

[22] S. K. Sarin and L. Lynch, Discarding obsolete information in a replicated data base system, *IEEE Transactions on Software Engineering*, **13**(1), 1987, 39–46.

[23] F. Schmuck, The Use of Efficient Broadcast in Asynchronous Distributed Systems, Ph. D. Thesis, Cornell University, TR88-928, 1988.

[24] M. Singhal, A heuristically-aided mutual exclusion algorithm for distributed systems, *IEEE Transactions on Computers*, **38**(5), 1989, 651–662.

[25] M. Singhal and A. Kshemkalyani, An efficient implementation of vector clocks, *Information Processing Letters*, **43**, August, 1992, 47–52.

[26] R. E. Strom and S. Yemini, Optimistic recovery in distributed systems, *ACM Transactions on Computer Systems*, **3**(3), 1985, 204–226.

[27] B. Sundararaman, U. Buy, and A. D. Kshemkalyani, Clock synchronization in wireless sensor networks: a survey, *Ad-Hoc Networks*, **3**(3), 2005, 281–323.
[28] G. T. J. Wuu and A. J. Bernstein, Efficient solutions to the replicated log and dictionary problems, *Proceedings of 3rd ACM Symposium on PODC*, 1984, 233–242.