Time and State in Distributed Systems

Vijay K. Garg[*]
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

Neeraj Mittal
Dept. of Computer Science
The University of Texas at Dallas
Richardson, TX 75083-0688, USA
neerajm@utdallas.edu

Abstract

We discuss two fundamental problems that arise in distributed systems. First, how to determine the order in which various events were executed. Second, how to obtain a consistent view of the system. To address the first problem, we describe different schemes that implement an abstract notion of time and can be used to order events in a distributed system. To address the second problem, we discuss ways to obtain a consistent state of the system possibly satisfying certain desirable property.

1 Introduction

A distributed system is characterized by multiple processes that are spatially separated and are running independently. As processes run, they change their *states* by executing *events*. Processes communicate with each other by exchanging messages over a set of communication channels. However, message delays are arbitrary and may be unbounded.

Two inherent limitations of distributed systems are: *lack of global clock* and *lack of shared memory*. This has two important implications. First, due to the absence of any system-wide clock that is equally accessible to all processes, the notion of common time does not exist in a distributed system; different processes may have different notions of time. As a result, it is not always possible to determine the order in which two events on different processes were executed. Second, since processes in a distributed system do not share common memory, it is not possible for an individual process to obtain an up-to-date state of the entire system. Further, because of the absence of a global clock, obtaining a meaningful state of the system, in which states of different processes are consistent with each other, is difficult.

We describe different schemes that implement an abstract notion of time and can be used to order events in a distributed system. We also discuss ways to obtain a consistent state of the system possibly satisfying certain desirable property.

2 Clocks and Ordering of Events

For many distributed applications such as distributed scheduling and distributed mutual exclusion, it is important to determine the order in which various events were executed. If the system has a shared global clock, then timestamping each event with the global clock would be sufficient to determine the order. However, if such a clock is not available, then it becomes impossible to

[*]supported in part by the NSF Grants ECS-9907213, CCR-9988225, and an Engineering Foundation Fellowship.

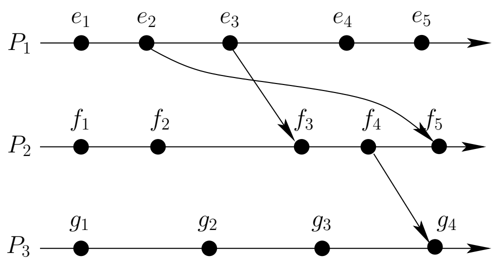


Figure 1: An example of a distributed computation.

determine the actual execution order of events. A natural question to ask is: what kind of ordering information can be ascertained in the absence of a global clock?

Each process in the system generates a sequence of events. Therefore it is clear how to order events within a single process. If event e occurred before f on a process, then e is ordered before f. But, how do we order events across processes? If e is the send event of a message and f is the receive event of the same message, then e is ordered before f. Combining these two ideas, we obtain the following definition:

Definition 1 (Happened-Before Relation) *The happened-before relation, denoted by $\rightarrow$, is the smallest transitive relation that satisfies the following:*

(1) If e occurred before f on the same process, then $e \rightarrow f$.

(2) If e is the send event of a message and f is the receive event of the same message, then $e \rightarrow f$.

As an example, consider a distributed computation involving three processes, namely P_1, P_2 and P_3, shown in Figure 1. In the figure, time progresses from left to right. Moreover, circles denote events and arrows between processes denote messages. Clearly, $e_2 \rightarrow e_4$, $e_3 \rightarrow f_3$, and $e_1 \rightarrow g_4$. Also, events e_2 and f_2 are not related by happened-before relation and therefore could have been executed in any order.

The concept of happened-before relation was proposed by Lamport [Lam78]. The happened-before relation imposes a *partial-order* on the set of events. Any extension of the happened-before relation to a total order gives a possible ordering in which events could have been executed.

For some distributed applications such as distributed mutual exclusion, it is sufficient to know *some total order* in which events *could have* been executed. The total order may or may not correspond to the actual order of execution of events. However, all processes must agree on the same total order. We next describe a mechanism to determine such an ordering at runtime.

2.1 Ordering Events Totally: Logical Clocks

A logical clock timestamps each event with an integer value such that the resulting order of events is consistent with the happened-before relation. Formally,

Definition 2 (Logical Clock) *A* logical clock *C is a map from the set of events E to the set of natural numbers $\mathcal{N}$ with the following constraint:*

$$\forall e, f \in E : e \rightarrow f \;\Rightarrow\; C(e) < C(f)$$

The implementation of logical clock, first proposed by Lamport [Lam78], uses an integer variable to simulate local clock on a process. On sending a message, the value of the local clock is incremented and then sent with the message. On receiving a message, a process takes the maximum of its own clock value and the value received with the message. After taking the maximum, the process increments the clock value. On executing an internal event, a process simply increments its clock. The algorithm can be used even when message communication is unreliable and unordered.

2.2 Ordering Events Partially: Vector Clocks

A logical clock establishes a total order on all events, even when two events are incomparable with respect to the happened-before relation. For many problems such as distributed debugging and distributed checkpointing and recovery, it is important to determine whether two given events are ordered using the happened-before relation or are incomparable.

The set of events E are partially ordered with respect to $\rightarrow$, but the domain of logical clock values, which is the set of natural numbers, is a total order with respect to $<$. Thus logical clocks do not provide complete information about the *happened-before* relation. We describe a mechanism called a *vector clock* that allows us to infer the happened-before relation completely.

Definition 3 (Vector Clock) *A* vector clock V *is a map from the set of events* E *to* $\mathcal{N}^N$ *(vectors of natural numbers) with the following constraint:*

$$\forall e, f \in E : e \rightarrow f \Leftrightarrow V(e) < V(f)$$

Because $\rightarrow$ is a partial order, it is clear that the timestamping mechanism should also result in a partial order. Thus the range of the timestamping function cannot be a total order like the set of natural numbers used for logical clocks. Instead, we use vectors of natural numbers. Given two vectors x and y of dimension N, we compare them as follows:

$$\begin{aligned} x \leq y &= (\forall k : 1 \leq k \leq N : x[k] \leq y[k]) \\ x < y &= (x \leq y) \vee (x \neq y) \end{aligned}$$

For example, $[1, 2, 1] < [2, 2, 3]$ but $[2, 3, 0]$ and $[0, 4, 1]$ are incomparable. The vector clock mechanism was proposed independently by Fidge [Fid91] and Mattern [Mat89]. Figure 2 shows an implementation of vector clock using vectors of size N, where N is the number of processes in the system.

The algorithm presented in Figure 2 is described by the initial conditions and the actions taken for each event type. A process increments its own component of the vector clock after each event (lines 4, 9 and 11). Furthermore, it includes a copy of its vector clock in every outgoing message (line 5). On receiving a message, it updates its vector clock by taking a componentwise maximum with the vector clock included in the message (lines 7 and 8). It is not required that message communication be ordered or reliable. A sample execution of the algorithm is given in Figure 3.

2.3 Higher-Dimensional Clocks

It is natural to ask whether two or more dimensional clocks can give processes additional knowledge. The answer is "yes." For an event e, let $e.v$ denote the value of the local clock immediately after executing e. A vector clock can be viewed as a knowledge vector. In this interpretation, for an event e on process P_k, $e.v[i]$ denotes what process P_k knows about process P_i after executing event e. In some applications it may be important for the process to have a still higher level of knowledge. The value $e.v[i, j]$ could represent what process P_k knows about what process P_i knows about process P_j. For example, if $e.v[i, k] \geq m$ for all i, then process P_k can conclude that everybody knows that it has executed at least m events.

```
Process $P_i$:
var
    $v$: array[1..$N$] of integer
        initially $(\forall j : j \neq i : v[j] = 0)$;

send event :
    $v[i] := v[i] + 1$;
    send $v$ along with the message;

receive event of a message tagged with vector $u$:
    for $j := 1$ to $N$ do
        $v[j] := \max(v[j], u[j])$;
    $v[i] := v[i] + 1$;

internal event :
    $v[i] := v[i] + 1$;
```

Figure 2: A vector clock algorithm.

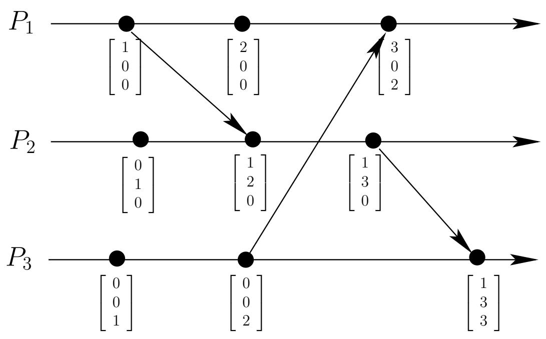

P_1
P_2
P_3
$\begin{bmatrix}1\\0\\0\end{bmatrix}$ $\begin{bmatrix}2\\0\\0\end{bmatrix}$ $\begin{bmatrix}3\\0\\2\end{bmatrix}$
$\begin{bmatrix}0\\1\\0\end{bmatrix}$ $\begin{bmatrix}1\\2\\0\end{bmatrix}$ $\begin{bmatrix}1\\3\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\end{bmatrix}$ $\begin{bmatrix}0\\0\\2\end{bmatrix}$ $\begin{bmatrix}1\\3\\3\end{bmatrix}$

Figure 3: A sample execution of the vector clock algorithm.

2.4 Physical Clocks

Until now, we assumed that message delays are arbitrary and unbounded. However, if message delays are bounded (but still arbitrary), then another way to timestamp events is to equip each process with a physical clock. Due to limitations in technology, it is possible for physical clocks on different processes to drift apart from each other. Therefore different physical clocks have to be synchronized with each other at a regular interval. Clocks are synchronized in a manner such that there exists a sufficiently small constant ϵ satisfying the following:

$$\forall i, j : |C_i(t) - C_j(t)| < \epsilon \tag{1}$$

where $C_i(t)$ denotes the value of the physical clock on process P_i at time t. Let $dC_i(t)/dt$ denote the rate at which the clock on process P_i is running at time t. Clearly, if C_i is an ideal clock, then $dC_i(t)/dt = 1$. Even if C_i is not an ideal clock, we assume that its rate of drift is bounded. Specifically, let κ be the maximum rate at which a clock can drift away from the actual time. Therefore for all i:

$$1 - \kappa < dC_i(t)/dt < 1 + \kappa \tag{2}$$

Clearly, to avoid anomalous behavior, the timestamp of a receive event should be greater than the timestamp of the corresponding send event. Therefore for all i, j and t:

$$C_i(t + \mu) > C_j(t) \tag{3}$$

where μ is the transmission time. To achieve synchronization of physical clocks that satisfies (1), (2) and (3), the following algorithm proposed by Lamport [Lam78] can be used:

(1) Each process sends a synchronization message to all its neighboring processes after every τ units of time. A process includes its value of local physical clock along with the message.

(2) A process, on receiving a synchronization message with timestamp T_m, sets its physical clock value to the maximum of its current value and $T_m + \mu_m$, where μ_m is the minimum amount of time required for message transmission.

3 Global State

To solve many problems in distributed systems such as termination detection, we need to examine the state of the entire system, which is also referred to as *global state* or *global snapshot*. (In contrast, state of a process is referred to as *local state* or *local snapshot*). A simple collection of local states, one from each process, may not correspond to a meaningful system state. To appreciate this, consider a distributed database for a banking application. Assume for simplicity that there are only two sites that keep the accounts for a customer. Also assume that the customer has \$500 at the first site and \$300 at the second site. In the absence of any communication between these sites, the total money of the customer can be easily computed to be \$800. However, if there is a transfer of \$200 from site A to site B, and a simple procedure is used to add up the accounts, we may falsely report that the customer has a total of \$1000 in his or her accounts (to the chagrin of the bank). This happens when the value at the first site is used before the transfer and the value at the second site after the transfer. Clearly, the two values are not consistent with each other. Note that \$1000 cannot be justified even by the messages in transit (or, that "the check is in the mail"). We now describe what it means for a global state to be meaningful or consistent.

3.1 Consistent Global State

Intuitively, a global state captures the set of events that have been executed so far. For a global state G to be consistent, it should satisfy the following condition:

$$\forall e, f : (e \rightarrow f) \wedge (f \in G) \Rightarrow e \in G$$

Sometimes, it is more convenient to describe a global state in terms of local states instead of events. For a local state s, let $s.p$ denote the process to which s belongs. We can extend the definition of the happened-before relation, which was defined on events, to local states as follows: $s \rightarrow t$ if $s.p$ executed an event e after s and $t.p$ executed an event f before t such that either $e = f$ or $e \rightarrow f$. Two local states s and t are *concurrent*, which is denoted by $s \parallel t$, if $s \not\rightarrow t$ and $t \not\rightarrow s$. For a global state G, let $G[i]$ refer to the local state of process P_i in G. We now define what it means for a global state to be consistent, when the global state is expressed as a collection of local states.

Definition 4 (Consistent Global State) *A global state G is consistent if it satisfies:*

$$\forall i, j : G[i] \parallel G[j]$$

In general, a global state can be used to deduce meaningful conclusions about the state of the system only if it is consistent.

3.2 Finding a Consistent Global State

We discuss how to obtain a consistent view of the entire system. The algorithm, which was proposed by Chandy and Lamport [CL85], assumes that all channels are unidirectional and satisfy the first-in-first-out (FIFO) property. Moreover, it also records the state of all communication channels, which is given by the set of messages in transit. The computation of the snapshot is initiated by one or more processes. We associate with each process a variable called *color* that is either white or red. All processes are initially white. Intuitively, the computed global snapshot corresponds to the state of the system just before processes turn red. After recording its local state, a process turns red. Thus the local snapshot of a process is simply the state just before it turned red. The algorithm relies on a special message called a marker. The consistent global snapshot algorithm is given by the following rules:

(1) **(Turning Red Rule)** When a process records its local state, it turns from white to red. On turning red, it sends out a marker on every outgoing channel before sending any application message on that channel. It also starts recording messages on all incoming channels.

(2) **(Marker Receiving Rule)** On receiving a marker, a white process turns red. The process also stops recording messages along that channel.

A process has finished its local snapshot when it has received a marker on each of its incoming channel. The algorithm requires that a marker be sent along all channels. Thus it has an overhead of one message per channel in the system. We have not discussed how to combine local snapshots into a global snapshot. A simple method would be for all processes to send their local snapshots to a predetermined process.

3.3 Finding a Consistent Global State Satisfying the Given Property

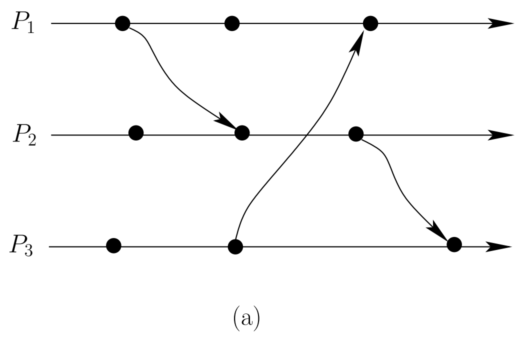
Sometimes, it is not sufficient to find just any consistent global state. Rather, we may want to find a consistent global state that satisfies certain global property [CM91, AV01]. If the global property is stable, that is, it stays true once it becomes true, then repeated invocations of the Chandy and Lamport's algorithm for taking a consistent global snapshot can be used to find the required global state.

We discuss an algorithm that can be used to find a consistent global state satisfying an unstable property. We will assume that the given global property, say, B, is constructed from local predicates using boolean connectives. We first show that B can be detected using an algorithm that can detect q, where q is a pure conjunction of local predicates. The predicate B can be rewritten in its disjunctive normal form as:

$$B = q_1 \vee \ldots \vee q_k \qquad\qquad k \geq 1$$

where each q_i is a pure conjunction of local predicates. Next, observe that a global state satisfies B if and only if it satisfies at least one of the q_i's. Thus the problem of detecting B is reduced to solving k problems of detecting q, where q is a pure conjunction of local predicates.

Formally, we define a *weak conjunctive predicate (WCP)* to be true for a given computation if and only if there exists a consistent global state in the computation for which all conjuncts are true [GW94]. Intuitively, detecting a weak conjunctive predicate is generally useful when one is interested in detecting a combination of states that is unsafe. For example, violation of mutual exclusion for a two-process system can be written as "P_1 is in the critical section and P_2 is in the critical section." To detect a weak conjunctive predicate, it is necessary and sufficient to find a



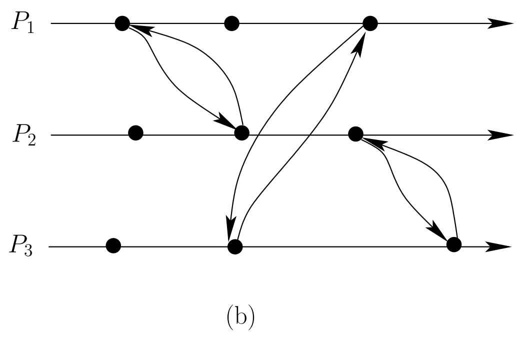


Figure 4: (a) A distributed computation, and (b) it slice with respect to the property "all channels are empty".

set of concurrent local states, one on each process, in which all local predicates are true. We now present an algorithm to do so.

In this algorithm, one process serves as a checker. All other processes involved in detecting the WCP are referred to as *application processes*. Each application process maintains a vector clock. It also checks for the respective local predicate. Whenever the local predicate of a process becomes true for the *first* time since the most recently sent message (or the beginning of the trace), it generates a debug message containing its local timestamp vector and sends it to the checker process.

Note that a process is not required to send its vector clock every time the local predicate is detected. If two local states, say, s and t, on the same process are separated only by internal events, then they are indistinguishable to other processes so far as consistency is concerned, that is, if u is a local state on some other process, then $s \parallel u$ if and only if $t \parallel u$. Thus it is sufficient to consider at most one local state between two external events and the vector clock need not be sent if there has been no message activity since the last time the vector clock was sent.

The checker process is responsible for searching for a consistent global state that satisfies the WCP by considering a sequence of candidate global states. If the candidate global state either is not consistent or does not satisfy some term of the WCP, the checker can efficiently eliminate one of the local states in the global state. The eliminated state can never be part of a consistent global state that satisfies the WCP. The checker can then advance the global state by considering the successor to one of the eliminated states. If the checker finds a global state for which no state can be eliminated, then that global state satisfies the WCP and the detection algorithm halts.

3.4 Finding All Consistent Global States Satisfying the Given Property

In debugging applications, it is sometimes useful to record all consistent global states that satisfy the given property. A computation slice is a concise representation of all such global states. A slice of a distributed computation with respect to a given property B is a concise representation of all the global states that satisfy B [MG01, SG03].

To understand the principle behind slicing, one needs to note that a computation (an acyclic directed graph on set of events) can be viewed as a generator of all consistent global states. A subset of vertices, H, of a directed graph is a *consistent global state* if it satisfies the following condition: if H contains a vertex v and (u, v) is an edge in the graph, then H also contains u. Given a computation, if one adds additional edges to the computation, the number of consistent possible global state can only decrease. The goal of slicing is to determine the maximum set of edges to add to the graph such that the resulting graph continues to contain all consistent global

```
1   graph function computeSlice($B$:boolean_predicate, $P$: graph)
2   var
3       $R$: graph initialized to $P$;
4   begin
5       for every pair of nodes $e$, $f$ in $P$ do
6           $Q := P$ with the additional edges $(f, \perp)$ and $(\top, e)$;
7           if $detect(B, Q)$ is false
8               add edge $(e, f)$ to $R$;
9       endfor
10      return $R$;
11  end;
```

Figure 5: An efficient algorithm to compute the slice for a predicate B

states of the computation that satisfy the given property. Note that when an edge is added to the original graph, the resulting graph may not be acyclic anymore.

As an example, consider the distributed computation shown in Figure 4(a). Its slice with respect to the global property "all channels are empty" is depicted in Figure 4(b).

There are three main motivations for computing all the global states that satisfy a given property. First, for debugging applications the programmer may not know the exact condition under which a bug occurs, but only that whenever the bug occurs B is true. Therefore we have to record all global states that satisfy B. Based on slicing, one can provide a "fast-forward" utility in debuggers where the system only goes through global states satisfying B. The second motivation comes from detecting predicates of the form $B_1 \wedge B_2$ in which the programmer knows efficient detection algorithm for B_1 but not B_2. Instead of searching the set of all global states for a global state that satisfies $B_1 \wedge B_2$, slicing allows the programmer to restrict the search to only those global states that satisfy B_1. This set of global states may be exponentially smaller than the original set of global states. The final motivation comes from detecting nested temporal logic predicates. Currently, the only efficient method known to detect nested temporal logic predicates is based on slicing.

We now show how to efficiently compute slices for the predicates for which there exists an efficient detection algorithm. The slicing algorithm is shown in Figure 5. It takes as input a graph P and a boolean predicate B. The input graph is obtained from the input computation by adding two additional vertices $\perp$ and $\top$ such that there is a path from $\perp$ to every vertex and a path from every vertex to $\top$. We refer to global states that contain $\perp$ but do not contain $\top$ as *nontrivial global states*.

The algorithm constructs the slice by adding edges to the graph P. For this purpose, it initializes in line 3 a graph R as P. In rest of the function, edges are added to R which is finally returned. The addition of edges is done as follows. For every pair of vertices, e and f, in the graph P, the algorithm constructs Q from P by adding edges from f to $\perp$ and $\top$ to e. Due to these edges in Q, all nontrivial global states of Q contain f and do not contain e. We now invoke the detection algorithm on Q. If the detection algorithm returns false, then we know that there is no nontrivial global states of P that contains f but does not contain e. Therefore, all global states that satisfy B have the property that if they include f, they also include e. Hence, we add an edge from e to f in the graph R. We continue this procedure for all pairs of vertices.

References

[AV01] S. Alagar and S. Venkatesan. Techniques to Tackle State Explosion in Global Predicate Detection. *IEEE Transactions on Software Engineering*, 27(8):704–714, August 2001.

[CL85] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM91] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.

[Fid91] C. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

[GW94] V. K. Garg and B. Waldecker. Detection of Weak Unstable Predicates in Distributed Programs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 5(3):299–307, March 1994.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[Mat89] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989.

[MG01] N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 78–92, October 2001.

[SG03] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates in Distributed Programs using Computation Slicing. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, December 2003.