

Time and Global States- Introduction-Clocks, events and process states-Synchronizing physical clocks-Logical time and logical clocks-Global states

Time and Global States

There are two formal models of distributed systems: synchronous and asynchronous.

Synchronous distributed systems have the following characteristics:

- the time to execute each step of a process has known lower and upper bounds;
- each message transmitted over a channel is received within a known bounded time;
- Each process has a local clock whose drift rate from real time has a known bound.

Asynchronous distributed systems, in contrast, guarantee no bounds on process execution speeds, message transmission delays, or clock drift rates. Most distributed systems we discuss, including the Internet, are asynchronous systems.

Generally, timing is a challenging and important issue in building distributed systems. Consider a couple of examples:

- Suppose we want to build a distributed system to track the battery usage of a bunch of laptop computers and we'd like to record the percentage of the battery each has remaining at exactly 2pm.
- Suppose we want to build a distributed, real time auction and we want to know which of two bidders submitted their bid first.
- Suppose we want to debug a distributed system and we want to know whether variable x_1 in process p_1 ever differs by more than 50 from variable x_2 in process p_2 .

In the first example, we would really like to synchronize the clocks of all participating computers and take a measurement of absolute time. In the second and third examples, knowing the absolute time is not as crucial as knowing the order in which events occurred.



Every computer has a physical clock that counts oscillations of a crystal. This hardware clock is used by the computer's software clock to track the current time. However, the hardware clock is subject to *drift* -- the clock's frequency varies and the time becomes inaccurate. As a result, any two clocks are likely to be slightly different at any given time. The difference between two clocks is called their *skew*.

There are several methods for synchronizing physical clocks. *External synchronization* means that all computers in the system are synchronized with an external source of time (e.g., a UTC signal). *Internal synchronization* means that all computers in the system are synchronized with one another, but the time is not necessarily accurate with respect to UTC.

In a synchronous system, synchronization is straightforward since upper and lower bounds on the transmission time for a message are known. One process sends a message to another process indicating its current time, t . The second process sets its clock to $t + (max+min)/2$ where max and min are the upper and lower bounds for the message transmission time respectively. This guarantees that the skew is at most $(max-min)/2$.

Cristian's method for synchronization in asynchronous systems is similar, but does not rely on a predetermined max and min transmission time. Instead, a process p_1 requests the current time from another process p_2 and measures the RTT (T_{round}) of the request/reply.

When p_1 receives the time t from p_2 it sets its time to $t + T_{round}/2$.

The Berkeley algorithm, developed for collections of computers running Berkeley UNIX, is an internal synchronization mechanism that works by electing a master to coordinate the synchronization. The master polls the other computers (called slaves) for their times, computes an average, and tells each computer by how much it should adjust its clock.

The Network Time Protocol (NTP) is yet another method for synchronizing clocks that uses a hierarchical architecture where the top level of the hierarchy (stratum 1) are servers connected to a UTC time source.

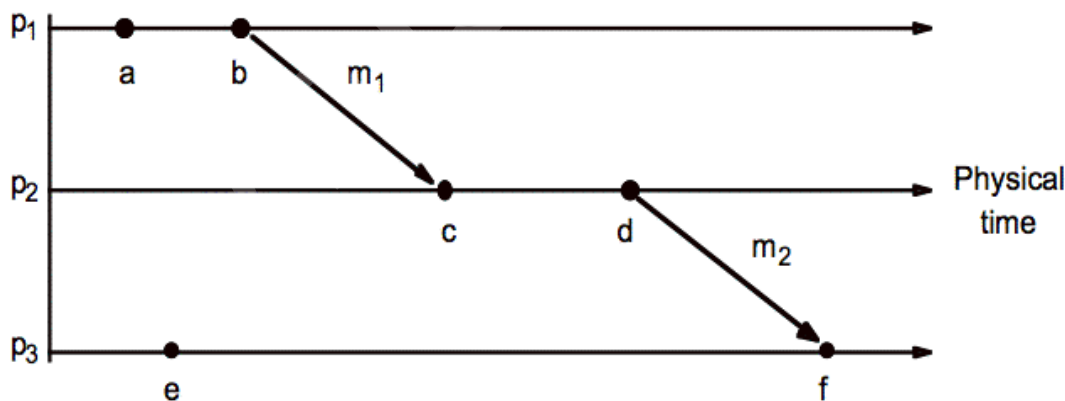
Logical Time

Physical time cannot be perfectly synchronized. Logical time provides a mechanism to define the *causal order* in which events occur at different processes. The ordering is based on the following:

- Two events occurring at the same process happen in the order in which they are observed by the process.
- If a message is sent from one process to another, the sending of the message happened before the receiving of the message.
- If e occurred before e' and e' occurred before e'' then e occurred before e'' .

"Lamport called the partial ordering obtained by generalizing these two relationships the *happened-before* relation." ($\rightarrow$)

Figure 11.5
Events occurring at three processes



In the figure, $a \rightarrow b$ and $c \rightarrow d$. Also, $b \rightarrow c$ and $d \rightarrow f$, which means that $a \rightarrow f$. However, we cannot say that $a \rightarrow e$ or vice versa; we say that they are *concurrent*

$(a \parallel e)$.

A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process p_i keeps its own logical clock, L_i , which it uses to apply so-called Lamport timestamps to events.

Lamport clocks work as follows:

LC1: L_i is incremented before each event is issued at p_i .

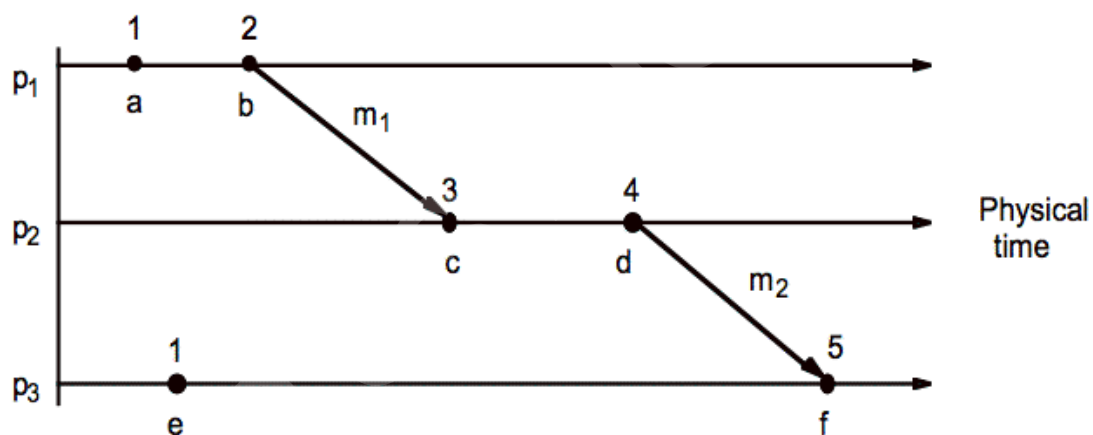
LC2:

When a process p_i sends a message m , it piggybacks on m the value $t = L_i$.

On receiving (m, t) , a process p_j computes $L_j := \max(L_j, t)$ and then applies LC1 before time stamping the event receive (m) .

An example is shown below:

Figure 11.6
Lamport timestamps for the events shown in Figure 11.5



If $e \rightarrow e'$ then $L(e) < L(e')$, but the converse is not true. Vector clocks address this problem.

"A vector clock for a system of N processes is an array of N integers." Vector clocks are updated as follows:

VC1: Initially, $V_i[j] = 0$ for $i, j = 1, 2, N$

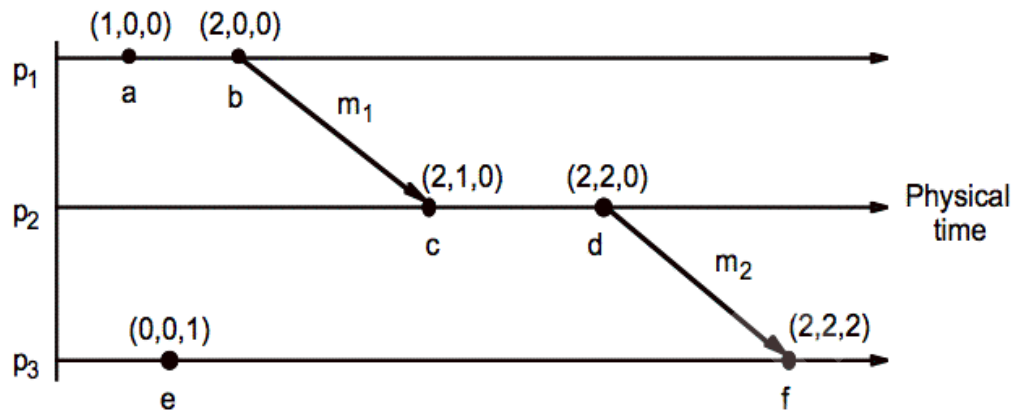
VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$.

VC3: p_i includes the value $t = V_i$ in every message it sends.

VC4: When p_i receives a timestamp t in a message, it sets $V_i[j] := \max(V_i[j], t[j])$, for $1, 2, \dots, N$. Taking the component wise maximum of two vector timestamps in this way is known as a merge operation.

An example is shown below:

Figure 11.7
Vector timestamps for the events shown in Figure 11.5



Vector timestamps are compared as follows:

$V = V'$ iff $V[j] = V'[j]$ for $j = 1, 2, \dots, N$

$V \leq V'$ iff $V[j] \leq V'[j]$ for $j = 1, 2, \dots, N$

$V < V'$ iff $V \leq V'$ and $V \neq V'$

If $e \rightarrow e'$ then $V(e) < V(e')$ and if $V(e) < V(e')$ then $e \rightarrow e'$.

Global States

It is often desirable to determine whether a particular property is true of a distributed system as it executes. We'd like to use logical time to construct a global view of the system state and determine whether a particular property is true. A few examples are as follows:

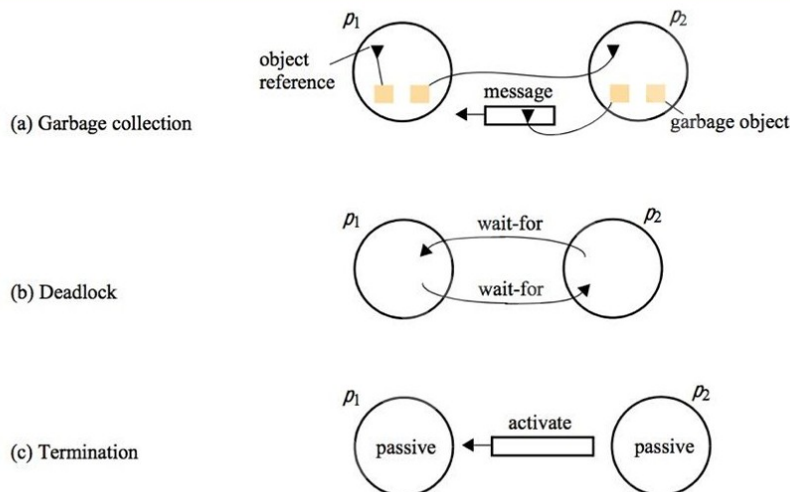
- Distributed garbage collection: Are there references to an object anywhere in the system? References may exist at the local process, at another process, or in the communication channel.
- Distributed deadlock detection: Is there a cycle in the graph of the "waits for" relationship between processes?
- Distributed termination detection: Has a distributed algorithm terminated?

- Distributed debugging: Example: given two processes p_1 and p_2 with variables x_1 and x_2 respectively, can we determine whether the condition $|x_1 - x_2| > \delta$ is ever true.

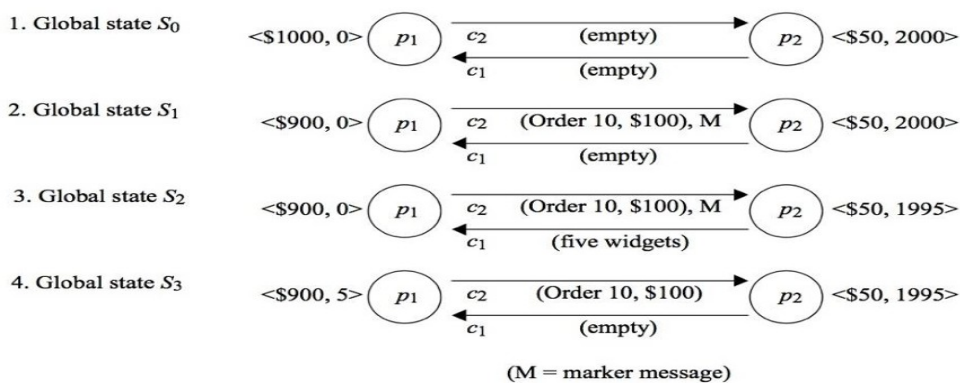
In general, this problem is referred to as *Global Predicate Evaluation*. "A global state predicate is a function that maps from the set of global state of processes in the system ρ to $\{\text{True}, \text{False}\}$."

- Safety - a predicate always evaluates to false. A given undesirable property (e.g., deadlock) never occurs.
- Liveness - a predicate eventually evaluates to true. A given desirable property (e.g., termination) eventually occurs.

Detecting global properties



The execution of the processes in Figure 14.11



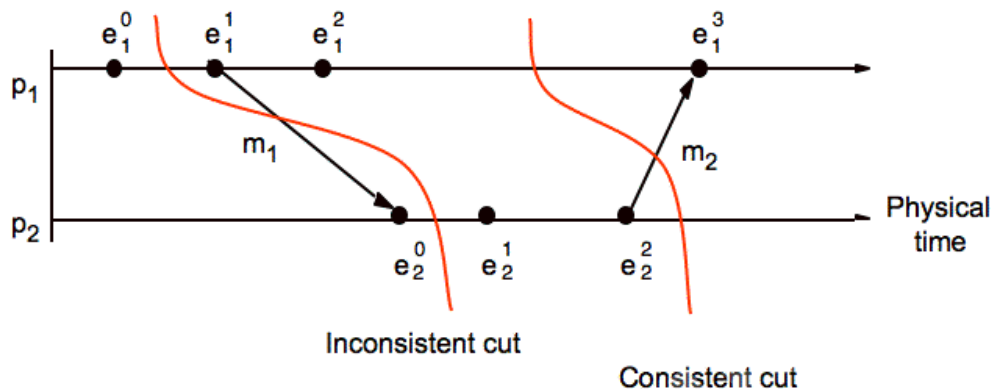
Cuts

Because physical time cannot be perfectly synchronized in a distributed system it is not possible to gather the global state of the system at a particular time. Cuts provide the ability to "assemble a meaningful global state from local states recorded at different times".

Definitions:

- ρ is a system of N processes p_i ($i = 1, 2, \dots, N$)
- $\text{history}(p_i) = h_i = \langle e_{i0}, e_{i1}, \dots \rangle$
- $h_{ik} = \langle e_{i0}, e_{i1}, \dots, e_{ik} \rangle$ - a finite prefix of the process's history
- s_{ik} is the state of the process p_i immediately before the k th event occurs
- All processes record sending and receiving of messages. If a process p_i records the sending of message m to process p_j and p_j has not recorded receipt of the message, then m is part of the state of the channel between p_i and p_j .
- A *global history* of ρ is the union of the individual process histories: $H = h_0 \cup h_1 \cup h_2 \cup \dots \cup h_{N-1}$
- A *global state* can be formed by taking the set of states of the individual processes: $S = (s_1, s_2, \dots, s_N)$
- A *cut* of the system's execution is a subset of its global history that is a union of prefixes of process histories (see figure below).
- The *frontier* of the cut is the last state in each process.
- A cut is *consistent* if, for all events e and e' :
 - o $(e \in C \text{ and } e' \rightarrow e) \Rightarrow e' \in C$
- A *consistent global state* is one that corresponds to a consistent cut.

Figure 11.9
 Cuts



Distributed Debugging

To further examine how you might produce consistent cuts, we'll use the distributed debugging example. Recall that we have several processes, each with a variable x_i . "The safety condition required in this example is $|x_i - x_j| \leq \delta$ ($i, j = 1, 2, \dots, N$)."

The algorithm we'll discuss is a centralized algorithm that determines post hoc whether the safety condition was ever violated. The processes in the system, $p_1, p_2, \dots, p_N$, send their states to a passive monitoring process, p_0 . p_0 is not part of the system. Based on the states collected, p_0 can evaluate the safety condition.

Collecting the state: The processes send their initial state to a monitoring process and send updates whenever relevant state changes, in this case the variable x_i . In addition, the processes need only send the value of x_i and a vector timestamp. The monitoring process maintains an ordered queue (by the vector timestamps) for each process where it stores the state messages. It can then create consistent global states which it uses to evaluate the safety condition.



Sri

SAIRAM

Regulation 2022(CBCS Scheme)

BCS515D- Distributed Systems

Chandy and Lamport's 'snapshot' algorithm

Marker receiving rule for process p_i

On p_i 's receipt of a *marker* message over channel c :

if (p_i has not yet recorded its state) *it*

records its process state now;

records the state of c as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of c as the set of messages it has received over c since it saved its state.

end if

Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel c :

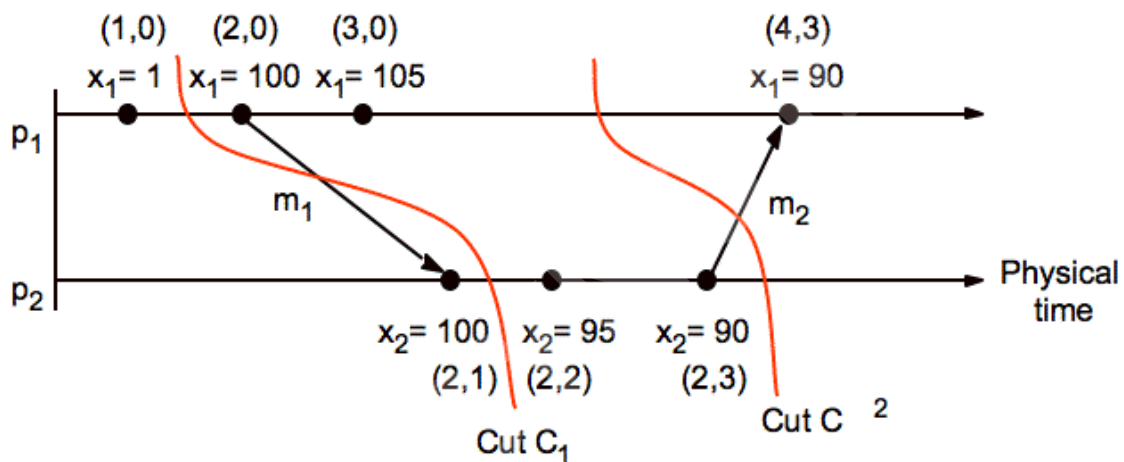
p_i sends one marker message over c

(before it sends any other message over c).

Let $S = (s_1, s_2, \dots, s_N)$ be a global state drawn from the state messages that the monitor process has received. Let $V(s_i)$ be the vector timestamp of the state s_i received from p_i . Then it can be shown that S is a consistent global state if and only if:

$$V(s_i)[i] \geq V(s_j)[i] \text{ for } i, j = 1, 2, \dots, N$$

Figure 11.14
Vector timestamps and variable values for the execution of Figure 11.9



Algorithms to evaluate *possibly* and *definitely*

1. Evaluating *possibly* ϕ for global history H of N processes

```

L := 0;
States := { (s1^0, s2^0, ..., sN^0) };
while (ϕ(S) = False for all S ∈ States)
    L := L + 1;
    Reachable := { S' : S' reachable in H from some S ∈ States ∧ level(S') = L };
    States := Reachable
end while
output "possibly ϕ";

```

2. Evaluating *definitely* ϕ for global history H of N processes

```

L := 0;
if (ϕ(s1^0, s2^0, ..., sN^0)) then States := {} else States := { (s1^0, s2^0, ..., sN^0) };
while (States ≠ {})
    L := L + 1;
    Reachable := { S' : S' reachable in H from some S ∈ States ∧ level(S') = L };
    States := { S ∈ Reachable : ϕ(S) = False }
end while
output "definitely ϕ";

```

