A Model of Distributed Computations

1.8 A Distributed Program

A distributed program is composed of a set of n asynchronous processes p_1, p_2…, p_i… ,p_n that communicate by message passing over the communication network. Without loss of generality, we assume that each process is running on a different processor. The processes do not share a global memory and communicate solely by passing messages.

Let C_{ij} denote the channel from process pi to process p_j and let m_{ij} denote a message sent by pi to p_j . The communication delay is finite and unpredictable. Also, these processes do not share a global clock that is instantaneously accessible to these processes. Process execution and message transfer are asynchronous – a process may execute an action spontaneously and a process sending a message does not wait for the delivery of the message to be complete.

The global state of a distributed computation is composed of the states of the processes and the communication channels. The state of a process is characterized by the state of its local memory and depends upon the context. The state of a channel is characterized by the set of messages in transit in the channel.

1.9 A model of distributed executions

The execution of a process consists of a sequential execution of its actions. The actions are atomic and the actions of a process are modeled as three types of events, namely, internal events, message send events, and message receive events. Let e_x^i denote the xth event at process p_i. Subscripts and/or superscripts will be dropped when they are irrelevant or are clear from the context. For a message m, let *send(m)* and *rec(m)* denote its send and receive events, respectively.

The occurrence of events changes the states of respective processes and channels, thus causing transitions in the global system state. An internal event changes the state of the process at which it occurs. A send event (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received). An internal event only affects the process at which it occurs.

The events at a process are linearly ordered by their order of occurrence. The execution of process p_i produces a sequence of events $e_i^1, e_i^2, ...e_i^x, ...e_i^{x+1}, ...$ and it is denoted by H_i,

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

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

The send and the receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process. A relation $\rightarrow_{msg}$ that captures the causal dependency due to message exchange, is defined as follows. For every message m that is exchanged between two processes, we have

$$send(m) \rightarrow_{msg} rec(m)$$

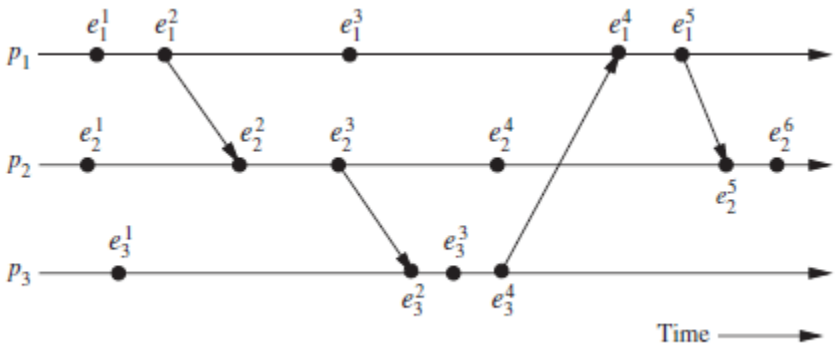

Figure: The space–time diagram of a distributed execution.

Relation $\rightarrow_{msg}$ defines causal dependencies between the pairs of corresponding send and receive events. The evolution of a distributed execution is depicted by a space–time diagram. Figure shows the space–time diagram of a distributed execution involving three processes. A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer.

Generally, the execution of an event takes a finite amount of time; however, since we assume that an event execution is atomic (hence, indivisible and instantaneous), it is justified to

denote it as a dot on a process line. In this figure, for process p_1, the second event is a message send event, the third event is an internal event, and the fourth event is a message receive event.

Causal precedence relation

The execution of a distributed application results in a set of distributed events produced by the processes. Let $H = \cup_i h_i$ denote the set of events executed in a distributed computation. Next, we define a binary relation on the set H, denoted as $\rightarrow$, that expresses causal dependencies between events in the distributed execution.

$$\forall e_i^x,\ \forall e_j^y \in H,\quad e_i^x \rightarrow e_j^y \quad \Leftrightarrow \quad \begin{cases} e_i^x \rightarrow_i e_j^y \text{ i.e., } (i = j) \wedge (x < y) \\ \text{or} \\ e_i^x \rightarrow_{msg} e_j^y \\ \text{or} \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \ \wedge \ e_k^z \rightarrow e_j^y \end{cases}$$

The causal precedence relation induces an irreflexive partial order on the events of a distributed computation that is denoted as H=(H, $\rightarrow$).

For any two events e_i and e_j , $e_i \not\rightarrow e_j$ denotes the fact that event e_j does not directly or transitively dependent on event e_i. That is, event e_i does not causally affect event e_j . Event e_j is not aware of the execution of e_i or any event executed after e_i on the same process.

Logical vs. physical concurrency

In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other. Physical concurrency, on the other hand, has a connotation that the events occur at the same instant in physical time. Note that two or more events may be logically concurrent even though they do not occur at the same instant in physical time.

Whether a set of logically concurrent events coincide in the physical time or in what order in the physical time they occur does not change the outcome of the computation. Therefore, even though a set of logically concurrent events may not have occurred at the same instant in physical time, for all practical and theoretical purposes, we can assume that these events occured at the same instant in physical time.