Chapter 2: A Model of Distributed Computations

A Distributed Program

- A distributed program is composed of a set of n asynchronous processes, p_1, p_2, ..., p_i, ..., p_n.
- The processes do not share a global memory and communicate solely by passing messages.
- The processes do not share a global clock that is instantaneously accessible to these processes.
- Process execution and message transfer are asynchronous.
- Without loss of generality, we assume that each process is running on a different processor.
- Let C_{ij} denote the channel from process p_i to process p_j and let m_{ij} denote a message sent by p_i to p_j.
- The message transmission delay is finite and unpredictable.

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_i^x denote the xth event at process p_i.
- 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.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that receives the message and the state of the channel on which the message is received.

A Model of Distributed Executions

- 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 is denoted by $\mathcal{H}_i$ where

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

  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.

A Model of Distributed Executions

- The evolution of a distributed execution is depicted by a space-time diagram.
- A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer.
- 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 the Figure 2.1, 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.

A Model of Distributed Executions

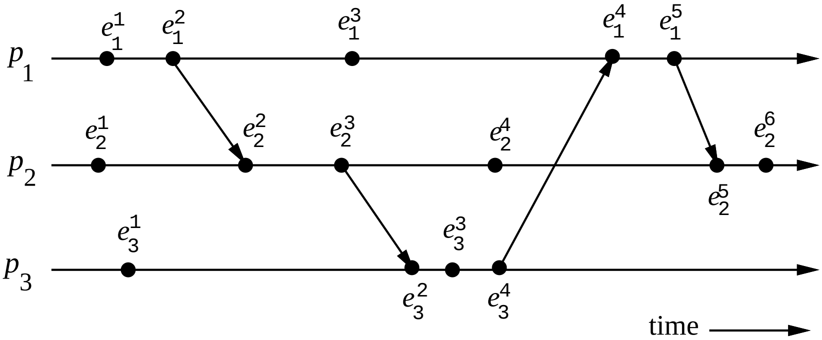


Figure 2.1: The space-time diagram of a distributed execution.

A Model of Distributed Executions

Concurrent events

- For any two events e_i and e_j, if $e_i \not\rightarrow e_j$ and $e_j \not\rightarrow e_i$, then events e_i and e_j are said to be concurrent (denoted as $e_i \parallel e_j$).
- In the execution of Figure 2.1, $e_1^3 \parallel e_3^3$ and $e_2^4 \parallel e_3^1$.
- The relation $\parallel$ is not transitive; that is, $(e_i \parallel e_j) \wedge (e_j \parallel e_k) \not\Rightarrow e_i \parallel e_k$.
- For example, in Figure 2.1, $e_3^3 \parallel e_2^4$ and $e_2^4 \parallel e_1^5$, however, $e_3^3 \not\parallel e_1^5$.
- For any two events e_i and e_j in a distributed execution, $e_i \rightarrow e_j$ or $e_j \rightarrow e_i$, or $e_i \parallel e_j$.

A Model of Distributed Executions

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.
- Two or more events may be logically concurrent even though they do not occur at the same instant in physical time.
- However, if processor speed and message delays would have been different, the execution of these events could have very well coincided in physical time.
- Whether a set of logically concurrent events coincide in the physical time or not, 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, we can assume that these events occured at the same instant in physical time.

Models of Communication Networks

- There are several models of the service provided by communication networks, namely, FIFO, Non-FIFO, and causal ordering.
- In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.

Models of Communication Networks

- The "causal ordering" model is based on Lamport's "happens before" relation.
- A system that supports the causal ordering model satisfies the following property:

  CO: For any two messages m_{ij} and m_{kj}, if $send(m_{ij}) \longrightarrow send(m_{kj})$, then $rec(m_{ij}) \longrightarrow rec(m_{kj})$.

- This property ensures that causally related messages destined to the same destination are delivered in an order that is consistent with their causality relation.
- Causally ordered delivery of messages implies FIFO message delivery. (Note that CO $\subset$ FIFO $\subset$ Non-FIFO.)
- Causal ordering model considerably simplifies the design of distributed algorithms because it provides a built-in synchronization.

Global State of a Distributed System

"A collection of the local states of its components, namely, the processes and the communication channels."

- The state of a process is defined by the contents of processor registers, stacks, local memory, etc. and depends on the local context of the distributed application.
- The state of channel is given by the set of messages in transit in the channel.
- The occurrence of events changes the states of respective processes and channels.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that or receives the message and the state of the channel on which the message is received.

. . . Global State of a Distributed System

A Consistent Global State

- Even if the state of all the components is not recorded at the same instant, such a state will be meaningful provided every message that is recorded as received is also recorded as sent.
- Basic idea is that a state should not violate causality – an effect should not be present without its cause. A message cannot be received if it was not sent.
- Such states are called *consistent global states* and are meaningful global states.
- Inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.
- A global state $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ is a *consistent global state* iff

$$\forall m_{ij} : \ send(m_{ij}) \not\leq LS_i^{x_i} \ \Leftrightarrow \ m_{ij} \not\in SC_{ij}^{x_i,y_j} \bigwedge rec(m_{ij}) \not\leq LS_j^{y_j}$$

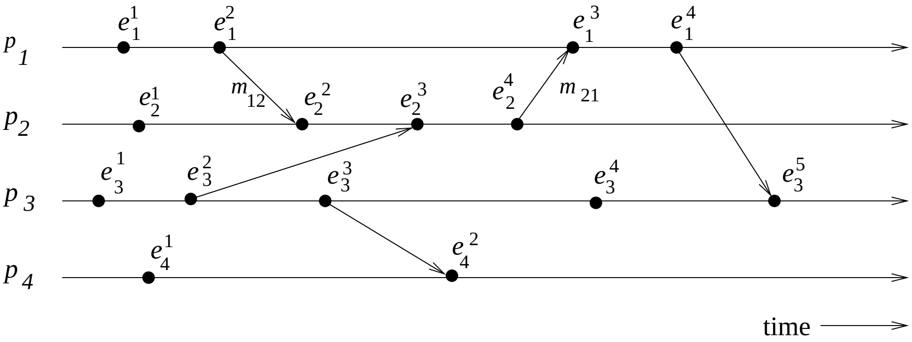
- That is, channel state $SC_{ij}^{y_i,z_k}$ and process state $LS_j^{z_k}$ must not include any message that process p_i sent after executing event $e_i^{x_i}$.

. . . Global State of a Distributed System

An Example

Consider the distributed execution of Figure 2.2.

Figure 2.2: The space-time diagram of a distributed execution.



. . . Global State of a Distributed System

In Figure 2.2:

- A global state $GS_1 = \{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is inconsistent because the state of p_2 has recorded the receipt of message m_{12}, however, the state of p_1 has not recorded its send.
- A global state GS_2 consisting of local states $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is consistent; all the channels are empty except C_{21} that contains message m_{21}.

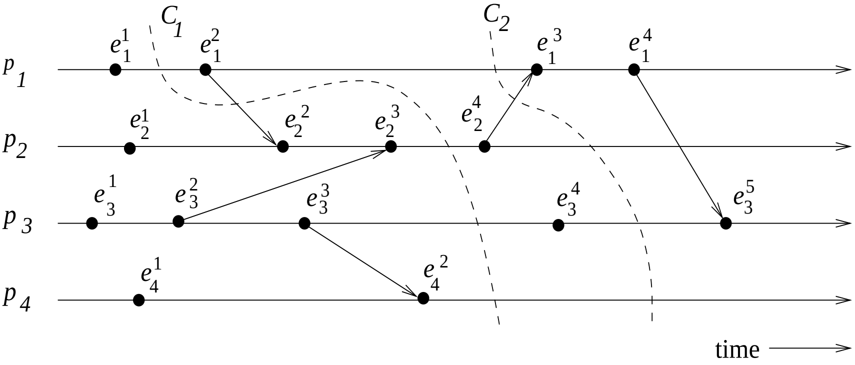
Cuts of a Distributed Computation

"In the space-time diagram of a distributed computation, a *cut* is a zigzag line joining one arbitrary point on each process line."

- A cut slices the space-time diagram, and thus the set of events in the distributed computation, into a PAST and a FUTURE.
- The PAST contains all the events to the left of the cut and the FUTURE contains all the events to the right of the cut.
- For a cut C, let PAST(C) and FUTURE(C) denote the set of events in the PAST and FUTURE of C, respectively.
- Every cut corresponds to a global state and every global state can be graphically represented as a cut in the computation's space-time diagram.
- Cuts in a space-time diagram provide a powerful graphical aid in representing and reasoning about global states of a computation.

... Cuts of a Distributed Computation

Figure 2.3: Illustration of cuts in a distributed execution.



Models of Process Communications

- There are two basic models of process communications – synchronous and asynchronous.
- The *synchronous* communication model is a blocking type where on a message send, the sender process blocks until the message has been received by the receiver process.
- The sender process resumes execution only after it learns that the receiver process has accepted the message.
- Thus, the sender and the receiver processes must synchronize to exchange a message. On the other hand,
- *asynchronous* communication model is a non-blocking type where the sender and the receiver do not synchronize to exchange a message.
- After having sent a message, the sender process does not wait for the message to be delivered to the receiver process.
- The message is bufferred by the system and is delivered to the receiver process when it is ready to accept the message.

. . . Models of Process Communications

- Neither of the communication models is superior to the other.
- Asynchronous communication provides higher parallelism because the sender process can execute while the message is in transit to the receiver.
- However, A buffer overflow may occur if a process sends a large number of messages in a burst to another process.
- Thus, an implementation of asynchronous communication requires more complex buffer management.
- In addition, due to higher degree of parallelism and non-determinism, it is much more difficult to design, verify, and implement distributed algorithms for asynchronous communications.
- Synchronous communication is simpler to handle and implement.
- However, due to frequent blocking, it is likely to have poor performance and is likely to be more prone to deadlocks.