Chapter 2: A Model of Distributed Computations

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press
A Distributed Program

- A distributed program is composed of a set of \( n \) asynchronous processes, \( p_1, p_2, \ldots, p_i, \ldots, p_n \).
- 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 $x$th 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 $e_{i1}^i, e_{i2}^i, ..., e_{ix}^i, e_{i(x+1)}^i, ...$ 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 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
  \[
  \text{send}(m) \rightarrow_{msg} \text{rec}(m).
  \]
- Relation \( \rightarrow_{msg} \) defines causal dependencies between the pairs of corresponding send and receive events.
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

Causal Precedence Relation

- The execution of a distributed application results in a set of distributed events produced by the processes.
- Let $H = \bigcup_i h_i$ denote the set of events executed in a distributed computation.
- Define a binary relation $\rightarrow$ on the set $H$ as follows that expresses causal dependencies between events in the distributed execution.

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

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

- Note that the relation $\rightarrow$ is nothing but Lamport’s “happens before” relation.
- For any two events $e_i$ and $e_j$, if $e_i \rightarrow e_j$, then event $e_j$ is directly or transitive dependent on event $e_i$. (Graphically, it means that there exists a path consisting of message arrows and process-line segments (along increasing time) in the space-time diagram that starts at $e_i$ and ends at $e_j$.)
- For example, in Figure 2.1, $e_1^1 \rightarrow e_3^3$ and $e_3^3 \rightarrow e_2^6$.
- The relation $\rightarrow$ denotes flow of information in a distributed computation and $e_i \rightarrow e_j$ dictates that all the information available at $e_i$ is potentially accessible at $e_j$.
- For example, in Figure 2.1, event $e_2^6$ has the knowledge of all other events shown in the figure.
A Model of Distributed Executions

...Causal Precedence Relation

- 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 $e_j$.
- In this case, event $e_j$ is not aware of the execution of $e_i$ or any event executed after $e_i$ on the same process.
- For example, in Figure 2.1, $e_3^1 \not\rightarrow e_3^2$ and $e_3^2 \not\rightarrow e_3^1$.

Note the following two rules:

- For any two events $e_i$ and $e_j$, $e_i \not\rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$.
- For any two events $e_i$ and $e_j$, $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$. 

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 $e_2^4 \parallel e_5^5$, however, $e_3^3 \parallel e_1^5$.
- 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 occurred 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 $send(m_{ij}) \rightarrow send(m_{kj})$, then $rec(m_{ij}) \rightarrow rec(m_{kj})$.

- 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

Notations

- \( LS^x_i \) denotes the state of process \( p_i \) after the occurrence of event \( e^x_i \) and before the event \( e^{x+1}_i \).
- \( LS^0_i \) denotes the initial state of process \( p_i \).
- \( LS^x_i \) is a result of the execution of all the events executed by process \( p_i \) till \( e^x_i \).
- Let \( send(m) \leq LS^x_i \) denote the fact that \( \exists y: 1 \leq y \leq x :: e^y_i = send(m) \).
- Let \( rec(m) \not\leq LS^x_i \) denote the fact that \( \forall y: 1 \leq y \leq x :: e^y_i \neq rec(m) \).
Global State of a Distributed System

A Channel State

- The state of a channel depends upon the states of the processes it connects.
- Let $SC_{ij}^{x,y}$ denote the state of a channel $C_{ij}$.

The state of a channel is defined as follows:

$$SC_{ij}^{x,y} = \{ m_{ij} \mid send(m_{ij}) \leq e_i^x \land rec(m_{ij}) \not\leq e_j^y \}$$

Thus, channel state $SC_{ij}^{x,y}$ denotes all messages that $p_i$ sent upto event $e_i^x$ and which process $p_j$ had not received until event $e_j^y$. 
Global State

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state $GS$ is defined as,
  \[ GS = \{ \bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k} \} \]
- For a global state to be meaningful, the states of all the components of the distributed system must be recorded at the same instant.
- This will be possible if the local clocks at processes were perfectly synchronized or if there were a global system clock that can be instantaneously read by the processes. (However, both are impossible.)
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 $GS = \{\bigcup_i LS_{i}^{x_i}, \bigcup_{j,k} SC_{j,k, y_j}^{z_k}\}$ is a consistent global state iff

$$\forall m_{ij} : \text{send}(m_{ij}) \not\leq LS_{i}^{x_i} \iff m_{ij} \not\in SC_{i,j}^{x_i, y_j} \land \text{rec}(m_{ij}) \not\leq LS_{j}^{y_j}$$

That is, channel state $SC_{i,j}^{y_j, z_k}$ 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.
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 \( \text{PAST}(C) \) and \( \text{FUTURE}(C) \) 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.
Cuts of a Distributed Computation

- In a consistent cut, every message received in the PAST of the cut was sent in the PAST of that cut. (In Figure 2.3, cut $C_2$ is a consistent cut.)
- All messages that cross the cut from the PAST to the FUTURE are in transit in the corresponding consistent global state.
- A cut is *inconsistent* if a message crosses the cut from the FUTURE to the PAST. (In Figure 2.3, cut $C_1$ is an inconsistent cut.)
Past and Future Cones of an Event

Past Cone of an Event

- An event $e_j$ could have been affected only by all events $e_i$ such that $e_i \rightarrow e_j$.
- In this situation, all the information available at $e_i$ could be made accessible at $e_j$.
- All such events $e_i$ belong to the past of $e_j$.

Let $\text{Past}(e_j)$ denote all events in the past of $e_j$ in a computation $(H, \rightarrow)$. Then,

$$\text{Past}(e_j) = \{ e_i | \forall e_i \in H, e_i \rightarrow e_j \}.$$

- Figure 2.4 (next slide) shows the past of an event $e_j$. 
...Past and Future Cones of an Event

Figure 2.4: Illustration of past and future cones.
Let $Past_i(e_j)$ be the set of all those events of $Past(e_j)$ that are on process $p_i$.

$Past_i(e_j)$ is a totally ordered set, ordered by the relation $\rightarrow_i$, whose maximal element is denoted by $max(Past_i(e_j))$.

$max(Past_i(e_j))$ is the latest event at process $p_i$ that affected event $e_j$ (Figure 2.4).
Let $\text{Max\_Past}(e_j) = \bigcup_{\forall i}\{\text{max}(\text{Past}_i(e_j))\}$.

$\text{Max\_Past}(e_j)$ consists of the latest event at every process that affected event $e_j$ and is referred to as the surface of the past cone of $e_j$.

Past$(e_j)$ represents all events on the past light cone that affect $e_j$.

**Future Cone of an Event**

The future of an event $e_j$, denoted by $\text{Future}(e_j)$, contains all events $e_i$ that are causally affected by $e_j$ (see Figure 2.4).

In a computation $(H, \rightarrow)$, $\text{Future}(e_j)$ is defined as:

$$\text{Future}(e_j) = \{ e_i | \forall e_i \in H, e_j \rightarrow e_i \}.$$
Define $Future_i(e_j)$ as the set of those events of $Future(e_j)$ that are on process $p_i$.

define $min(Future_i(e_j))$ as the first event on process $p_i$ that is affected by $e_j$.

Define $Min_Future(e_j)$ as $\bigcup_{i} \{ min(Future_i(e_j)) \}$, which consists of the first event at every process that is causally affected by event $e_j$.

$Min_Future(e_j)$ is referred to as the surface of the future cone of $e_j$.

All events at a process $p_i$ that occurred after $max(Past_i(e_j))$ but before $min(Future_i(e_j))$ are concurrent with $e_j$.

Therefore, all and only those events of computation $H$ that belong to the set “$H − Past(e_j) − Future(e_j)$” are concurrent with event $e_j$. 

...Past and Future Cones of an Event
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 buffered 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.