CHAPTER – 2

A MODEL OF DISTRIBUTED COMPUTATIONS

A distributed system consists of a set of processors that are connected by a communication network. The communication network provides the facility of information exchange among processors. The communication delay is finite but unpredictable.

The processors do not share a common global memory and communicate solely by passing messages over the communication network. There is no physical global clock in the system to which processes have instantaneous access.

The communication medium may deliver messages out of order, messages may be lost, garbled, or duplicated due to timeout and retransmission, processors may fail, and communication links may go down.

The system can be modeled as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels. A distributed application runs as a collection of processes on a distributed system.

2.1 A Distributed Program

A distributed program is composed of a set of n asynchronous processes **p1 , p2 , . . . , pi , . . . , pn** 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 $\mathbf{C_{ij}}$ denote the channel from process $\mathbf{p_i}$ to process $\mathbf{p_j}$ and let $\mathbf{m_{ij}}$ denote a message sent by $\mathbf{p_i}$ **to** $\mathbf{p_j}$.

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.

2.2 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 processes.

These processes are modeled as three types of events.

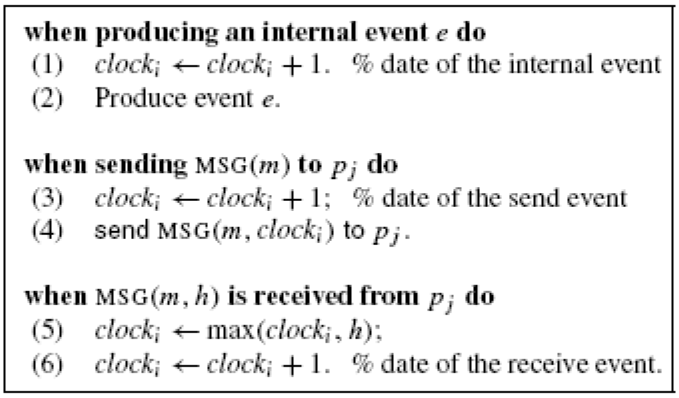
(1) Internal Events (2) Message Send Events (3) Message Receive Events

Let $\mathbf{e_i^x}$ denote the xth event at process $\mathbf{p_i}$. 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.

(1) Internal Events

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.



```
when producing an internal event $e$ do
(1)   $clock_i \leftarrow clock_i + 1$.  % date of the internal event
(2)   Produce event $e$.

when sending MSG($m$) to $p_j$ do
(3)   $clock_i \leftarrow clock_i + 1$;  % date of the send event
(4)   send MSG($m, clock_i$) to $p_j$.

when MSG($m, h$) is received from $p_j$ do
(5)   $clock_i \leftarrow \max(clock_i, h)$;
(6)   $clock_i \leftarrow clock_i + 1$.  % date of the receive event.
```

The events at a process are linearly ordered by their order of occurrence.

The execution of process $\mathbf{p_i}$ produces a sequence of events $\mathbf{e_i^1, e_i^2, \ldots\ e_i^x, e_i^{x+1}, \ldots}$ and is denoted by $\boldsymbol{H_i}$:

$$\boldsymbol{H_i} = (\mathbf{h_i}, \rightarrow_i)$$

where, $\mathbf{h_i}$ is the set of events produced by $\mathbf{p_i}$ and binary relation $\rightarrow_i$ defines a linear order on these events.

Relation $\rightarrow_i$ expresses causal dependencies among the events of $\mathbf{p_i}$.

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)**.

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.

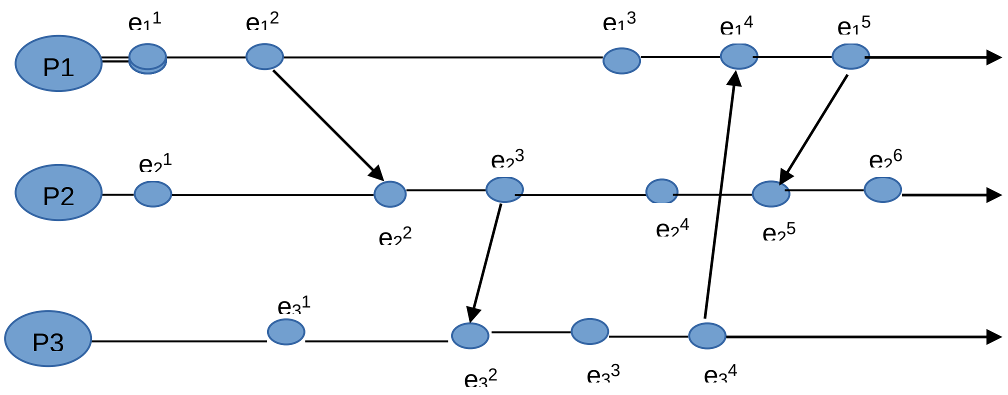


Figure 2.1 The space-time diagram of a distributed executions

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 $\mathbf{p_1}$, 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. The causal precedence relation between processes is captured by Lamport's "**happens before"** relation.

Let $\boldsymbol{H}=\cup_i \boldsymbol{h_i}$ denote the set of events executed in a distributed computation.

Define a binary relation → on the set $\boldsymbol{H}$ as follows that expresses causal dependencies between events in the distributed execution.

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

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

Note that the relation → is nothing but Lamport's "**happens before**" relation.

For any two events ei and ej , if ei → ej , then event ej is directly or transitively dependent on event ei . 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 ei and ends at ej.

From Figure 2.1, $\mathbf{e_1^1 \rightarrow e_3^3}$ **and** $\mathbf{e_3^3 \rightarrow e_2^6}$

The relation → denotes flow of information in a distributed computation and **ei → ej** dictates that all the information available at **ei** is potentially accessible at **ej** .

In Figure 2.1, event $\mathbf{e_2^6}$ has the knowledge of all other events shown in the figure.

For any two events **ei** and **ej** , $\mathbf{e_i^6}$**→ ej** denotes the fact that event ej does not directly or transitively dependent on event **ei .** That is, event **ei** does not causally affect event **ej.**

In this case, event ej is not aware of the execution of ei or any event executed after ei on the same process.

The representation of # is used for "**NOT**" in this whole chapter.

For example, in Figure 2.1, $\mathbf{e_1^3}$ **#→** $\mathbf{e_3^3}$ **and** $\mathbf{e_2^4}$ **#→** $\mathbf{e_3^1}$.

for any two events e_i and e_j , $\mathbf{e_i}$ **#→** $\mathbf{e_j}$ **⇒** $\mathbf{e_j}$ **#→** $\mathbf{e_i}$

for any two events e_i and e_j , $\mathbf{e_i}$ **#→** $\mathbf{e_j}$ **#⇒** $\mathbf{e_j}$ **#→** $\mathbf{e_i}$

For any two events e i and e j , if **e i #→ e j and e j #→ e i** , then events ei and ej are said to be concurrent and the relation is denoted as **ei || ej** .
In the execution of Figure 2.1, $e_1^3 \parallel e_3^3$ and $e_2^4 \parallel e_3^1$. Note that relation || is not transitive; that is, (ei || ej) ∧ (ej || ek) #⇒ ei || ek . 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 #|| e_1^5 .

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.

For example, in Figure 2.1, events in the set **{e_1^3 , e_2^4 , e_3^3 }** are logically concurrent, but they occurred at different instants in physical time.

However, if processor speed and message delays had 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 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 occurred at the same instant in physical time.

2.3 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.

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 mij and mkj ,

if **send(m_{ij}) → send(m_{kj}), then rec(m_{ij}) → 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 ⊂ FIFO ⊂ Non-FIFO.)

Causal ordering model considerably simplifies the design of distributed algorithms because it provides a built-in synchronization.

Advantage of Causal Ordering

It eliminates the need for checking to ensure database consistency in the replicated database system.

For example, every process responsible for updating a replica receives the updates in the same order to maintain database consistency in replicated database systems.

2.4 Global State of a Distributed System

The global state of a distributed system is a collection of the states of its processes and the state of communication channels connecting the processes.

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.

Notations

- $\mathbf{LS_i^x}$ denotes the state of process $\mathbf{p_i}$ after the occurrence of event $\mathbf{e_i^x}$ and before the event $\mathbf{e_i^{x+1}}$.
- $\mathbf{LS_i^0}$ denotes the initial state of process $\mathbf{p_i}$.
- $\mathbf{Ls_i^x}$ is a result of the execution of all the events executed by process p_i till $\mathbf{e_i^x}$.
- Let $\mathbf{send(m) \leq Ls_i^x}$ denotes the fact that $\mathbf{\exists y: 1 \leq y \leq x :: \ e_i^y = send(m)}$
- Let $\mathbf{rec(m) \# \leq Ls_i^x}$ denotes the fact that $\mathbf{\forall y: 1 \leq y \leq x :: \ e_i^y \# = rec(m)}$

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:

$$SC_{ij}^{x,y} = \{ m_{ij} \mid send(m_{ij}) \leq e_i^x \wedge rec(m_{ij}) \# \leq e_j^y \}$$

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

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,

$$GS = \{U_i LS_i^{Xi} , U_{j,k} SC_{jk}^{Yj,Zk} \}$$

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 **$GS = \{U_i LS_i^{Xi}, U_{j,k} SC_{jk}^{Yj,Zk}\}$** is a consistent global state iff

$$\forall m_{ij} : send(m_{ij}) \# \leq LS_i^{Xi} \Leftrightarrow m_{ij} \# \in Sc_{ij}^{Xj,Yj} \wedge rec(m_{ij}) \# \leq LS_j^{Yj}$$

That is, channel state **$SC_{jk}^{Yj,Zk}$** and process state **LS_j^{Zk}** must not include any message that process **p_i** sent after executing event **e_i^{Xi}**.

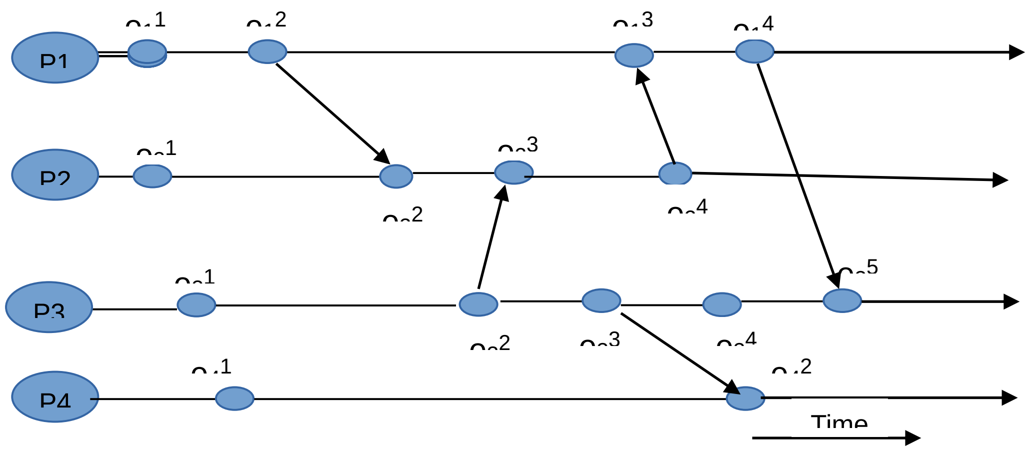


Figure 2.2 Space-time diagram of Distributed Executions

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}.

2.5 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 represents a global state in the distributed computation.

- 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.

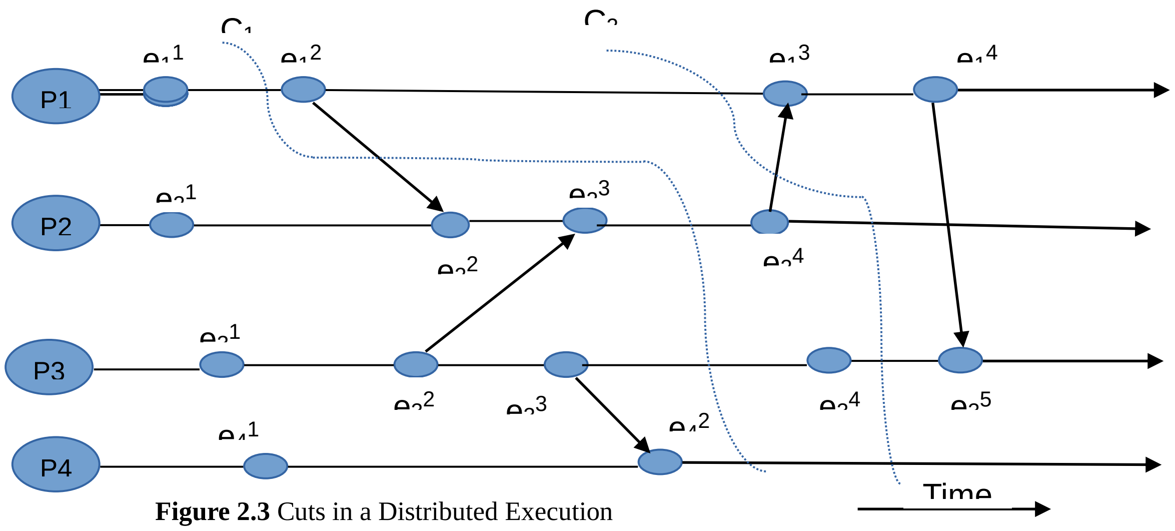


Figure 2.3 Cuts in a Distributed Execution

Cuts in a space–time diagram provide a powerful graphical aid in representing and reasoning about global states of a 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.)

2.6 Past Cones of an Event

The past of an event consists of all events that causally affect it.

An event $\mathbf{e_j}$ could have been affected only by all events $\mathbf{e_i}$ such that $\mathbf{e_i} \rightarrow \mathbf{e_j}$.

In this situtaion, all the information available at $\mathbf{e_i}$ could be made accessible at $\mathbf{e_j}$.

All such events $\mathbf{e_i}$ belong to the past of $\mathbf{e_j}$.

Let Past(e_j) denote all events in the past of $\mathbf{e_j}$ in a computation $(H, \rightarrow)$. Then, $\boldsymbol{Past(e_j) = \{e_i \mid \forall e_i \in H, e_i \rightarrow e_j\}}$.

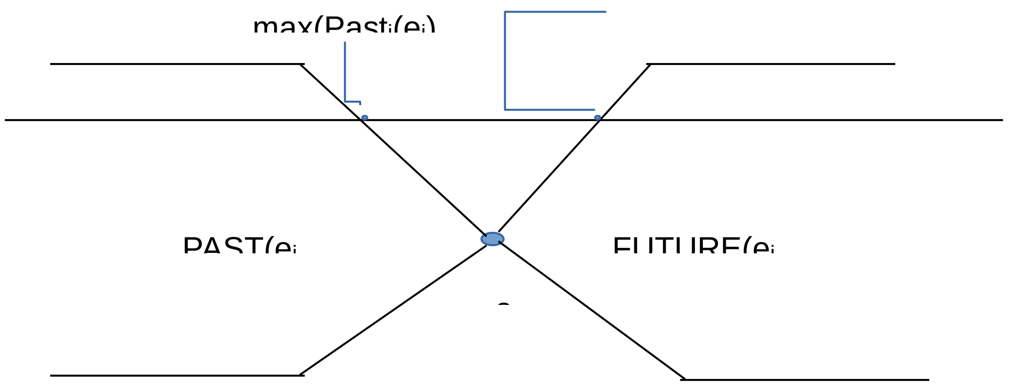


Figure 2.4 Illustration of Past and Future Cones

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 **Max *$Past(e_j)$* = $\cup_{(\forall i)}$ {max(*$Past(e_j)$*)}.**

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*** .

2.7 Future Cone of an Event

The future of an event consists of all events that are causally affected by it.

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

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

$$\textbf{Future}(e_j) = \{e_i \mid \forall e_i \in H,\ e_j \rightarrow e_i\}.$$

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 ***$Min\_Future_i(e_j)$*** as **$\cup_{(\forall i)}$ {*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*** .

2.8 Models of Process Communication

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.

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.

The ***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. A buffer overflow may occur if a process sends a large number of messages in a burst to another process.

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.

$$\forall m_{ij} : send(m_{ij}) \#\leq LS_i^{Xi} \Leftrightarrow m_{ij} \#\in Sc_{ij}^{Xj,Yj} \wedge rec(m_{ij}) \#\leq LS_j^{Yj}$$

That is, channel state **$SC_{jk}^{Yj,Zk}$** and process state **LS_j^{Zk}** must not include any message that process **p_i** sent after executing event **e_i^{Xi}**.