DISTRIBUTED COMPUTING
MODULE-1

Definition

A distributed system is a collection of independent computers that appears to its users as a single coherent system. A distributed system consists of components (i.e., computers) that are autonomous and users (be they people or programs) think they are dealing with a single system. This means that one way or the other the autonomous components need to collaborate.

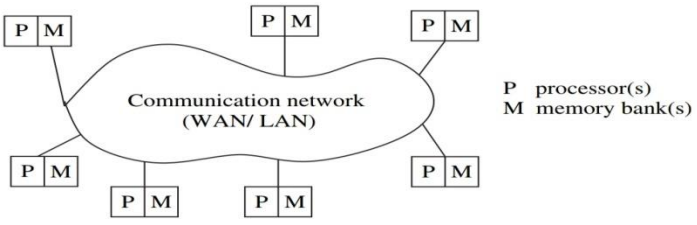
The computers, processes, or processors are referred to as the nodes of the distributed system. To be qualified as "autonomous" , the nodes must at least be equipped with their own private control; thus, a parallel computer of the single-instruction, multiple-data (SIMD) model does not qualify as a distributed system. To be qualified as "interconnected" , the nodes must be able to exchange information.

Characteristics of a distributed system:

1. **No common physical clock** This is an important assumption because it introduces the element of "distribution" in the system and gives rise to the inherent asynchrony amongst the processors.
2. **No shared memory** This is a key feature that requires message-passing for communication. This feature implies the absence of the common physical clock.
3. **Geographical separation** The geographically wider apart that the processors are, the more representative is the system of a distributed system. However, it is not necessary for the processors to be on a wide-area network (WAN). Recently, the network/cluster of workstations (NOW/COW) configuration connecting processors on a LAN is also being increasingly regarded as a small distributed system.
4. **Autonomy and heterogeneity** The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system. They are usually not part of a dedicated system, but cooperate with one another by offering services or solving a problem jointly.

Relation to computer system components

A typical distributed system is shown in Figure. Each computer has a memory-processing unit and the computers are connected by a communication network. The distributed software is also termed as *middleware*. A *distributed execution* is the execution of processes across the distributed system to collaboratively achieve a common goal. An execution is also sometimes termed a *computation* or a *run*.



The distributed system uses a layered architecture to break down the complexity of system design. The middleware is the distributed software that drives the distributed system, while providing transparency of heterogeneity at the platform level. The middleware layer does not contain the traditional application layer functions of the network protocol stack, such as *http*, *mail*, *ftp*, and *telnet*. Various primitives and calls to functions defined in various libraries of the middleware layer are embedded in the user program code.

Motivation

The motivation for using a distributed system is some or all of the following requirements:

1. **Inherently distributed computations** In many applications such as money transfer in banking, or reaching consensus among parties that are geographically distant, the computation is inherently distributed.
2. **Resource sharing** Resources such as peripherals, complete data sets in databases, special libraries, as well as data (variable/files) cannot be fully replicated at all the sites because it is often neither practical nor cost-effective. Further, they cannot be placed at a single site because access to that site might prove to be a bottleneck. Therefore, such resources are typically distributed across the system.
3. **Access to geographically remote data and resources** In many scenarios, the data cannot be replicated at every site participating in the distributed execution because it may be too large or too sensitive to be replicated. For example, payroll data within a multinational corporation is both too large and too sensitive to be replicated at every branch office/site. It is therefore stored at a central server which can be queried by branch offices. Similarly, special resources such as supercomputers exist only in certain locations, and to access such supercomputers, users need to log in remotely.
4. **Enhanced reliability** A distributed system has the inherent potential to provide increased reliability because of the possibility of replicating resources and executions, as well as the reality that geographically distributed resources are not likely to crash/malfunction at the same time under normal circumstances. Reliability entails several aspects:
   - availability, i.e., the resource should be accessible at all times;
   - integrity, i.e., the value/state of the resource should be correct, in the face of concurrent access from multiple processors, as per the semantics expected by the application;
   - fault-tolerance, i.e., the ability to recover from system failures, where such failures may be defined to occur in one of many failure models
5. **Scalability** As the processors are usually connected by a wide-area network, adding more processors does not pose a direct bottleneck for the communication network
6. **Increased performance/cost ratio** By resource sharing and accessing geographically remote data and resources, the performance/cost ratio is increased. Although higher throughput has not necessarily been the main objective behind using a distributed system, nevertheless, any task can be partitioned across the various computers in the distributed system. Such a configuration provides a better performance/cost ratio than using special parallel machines.

Primitives for distributed communication

Message send and message receive communication primitives are denoted *Send()* and *Receive()*, respectively. A *Send* primitive has at least two parameters – the destination, and the buffer in the user space, containing the data to be sent. Similarly, a *Receive* primitive has at least two parameters – the source from which the data is to be received (this could be a wildcard), and the user buffer into which the data is to be received.

SAMSUNG MANUAL

There are two ways of sending data when the *Send* primitive is invoked – the buffered option and the unbuffered option. The *buffered option* which is the standard option copies the data from the user buffer to the kernel buffer. The data later gets copied from the kernel buffer onto the network. In the *unbuffered option*, the data gets copied directly from the user buffer onto the network. For the *Receive* primitive, the buffered option is usually required because the data may already have arrived when the primitive is invoked, and needs a storage place in the kernel.

- ✓ **Synchronous primitives** A *Send* or a *Receive* primitive is *synchronous* if both the *Send()* and *Receive()* handshake with each other. The processing for the *Send* primitive completes only after the invoking processor learns that the other corresponding *Receive* primitive has also been invoked and that the receive operation has been completed. The processing for the *Receive* primitive completes when the data to be received is copied into the receiver's user buffer.
- ✓ **Asynchronous primitives** A *Send* primitive is said to be *asynchronous* if control returns back to the invoking process after the data item to be sent has been copied out of the user-specified buffer. It does not make sense to define asynchronous *Receive* primitives.
- ✓ **Blocking primitives** A primitive is *blocking* if control returns to the invoking process after the processing for the primitive (whether in synchronous or asynchronous mode) completes.
- ✓ **Non-blocking primitives** A primitive is *non-blocking* if control returns back to the invoking process immediately after invocation, even though the operation has not completed. For a non-blocking *Send*, control returns to the process even before the data is copied out of the user buffer.

  For a non-blocking *Receive*, control returns to the process even before the data may have arrived from the sender. For non-blocking primitives, a return parameter on the primitive call returns a system-generated *handle* which can be later used to check the status of completion of the call. The process can check for the completion of the call in two ways. First, it can keep checking (in a loop or periodically) if the handle has been flagged or *posted*. Second, it can issue a *Wait* with a list of handles as parameters. The *Wait* call usually blocks until one of the parameter handles is posted.
- ✓ **Blocking synchronous *Send*** The data gets copied from the user buffer to the kernel buffer and is then sent over the network. After the data is copied to the receiver's system buffer and a *Receive* call has been issued, an acknowledgement back to the sender causes control to return to the process that invoked the *Send* operation and completes the *Send*.
- ✓ **non-blocking synchronous *Send*** Control returns back to the invoking process as soon as the copy of data from the user buffer to the kernel buffer is initiated. A parameter in the non-blocking call also gets set with the handle of a location that the user process can later check for the completion of the synchronous send operation. The location gets posted after an acknowledgement returns from the receiver
- ✓ **Blocking asynchronous *Send*** The user process that invokes the *Send* is blocked until the data is copied from the user's buffer to the kernel buffer. (For the unbuffered option, the user process that invokes the *Send* is blocked until the data is copied from the user's buffer to the network.)
- ✓ **non-blocking asynchronous *Send*** The user process that invokes the *Send* is blocked until the transfer of the data from the user's buffer to the kernel buffer is initiated. Control returns to the user process as soon as this transfer is initiated, and a parameter in the non-blocking call also gets set with the handle of a location that the user process can check later

SAMPLE

using the *Wait* operation for the completion of the asynchronous *Send* operation. The asynchronous *Send* completes when the data has been copied out of the user's buffer.

- ✓ **Blocking *Receive*** The *Receive* call blocks until the data expected arrives and is written in the specified user buffer. Then control is returned to the user process.
- ✓ **non-blocking *Receive*** The *Receive* call will cause the kernel to register the call and return the handle of a location that the user process can later check for the completion of the non-blocking *Receive* operation. This location gets posted by the kernel after the expected data arrives and is copied to the user-specified buffer. The user process can check for the completion of the non-blocking *Receive* by invoking the *Wait* operation on the returned handle.

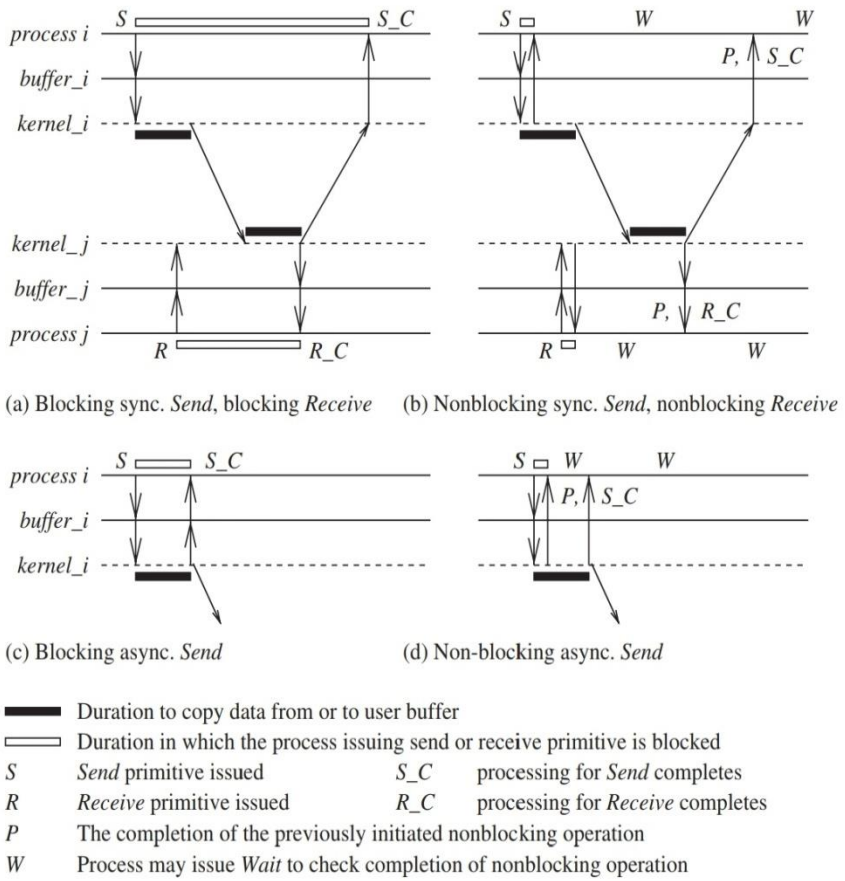


Figure 1.8 Blocking/non-blocking and synchronous/asynchronous primitives [12]. Process P_i is sending and process P_j is receiving. (a) Blocking synchronous *Send* and blocking (synchronous) *Receive*. (b) Non-blocking synchronous *Send* and nonblocking (synchronous) *Receive*. (c) Blocking asynchronous *Send*. (d) Non-blocking asynchronous *Send*.

- ✓ ***Processor synchrony*** indicates that all the processors execute in lock-step with their clocks synchronized. As this synchrony is not attainable in a distributed system, what is more generally indicated is that for a large granularity of code, usually termed as a *step*, the processors are synchronized. This abstraction is implemented using some form of barrier synchronization to ensure that no processor begins executing the next step of code until all the processors have completed executing the previous steps of code assigned to each of the processors.

Design issues and challenges in distributed systems

The following functions must be addressed when designing and building a distributed system:

Communication This task involves designing appropriate mechanisms for communication among the processes in the network. Some example mechanisms are: remote procedure call (RPC), remote object invocation (ROI), message-oriented communication versus stream-oriented communication.
Processes Some of the issues involved are: management of processes and threads at clients/servers; code migration; and the design of software and mobile agents.
Naming Devising easy to use and robust schemes for names, identifiers, and addresses is essential for locating resources and processes in a transparent and scalable manner. Naming in mobile systems provides additional challenges because naming cannot easily be tied to any static geographical topology.
Synchronization Mechanisms for synchronization or coordination among the processes are essential. Mutual exclusion is the classical example of synchronization, but many other forms of synchronization, such as leader election are also needed. In addition, synchronizing physical clocks, and devising logical clocks that capture the essence of the passage of time, as well as global state recording algorithms, all require different forms of synchronization.
Data storage and access Schemes for data storage, and implicitly for accessing the data in a fast and scalable manner across the network are important for efficiency. Traditional issues such as file system design have to be reconsidered in the setting of a distributed system.
Consistency and replication To avoid bottlenecks, to provide fast access to data, and to provide scalability, replication of data objects is highly desirable. This leads to issues of managing the replicas, and dealing with consistency among the replicas/caches in a distributed setting. A simple example issue is deciding the level of granularity (i.e., size) of data access.
Fault tolerance Fault tolerance requires maintaining correct and efficient operation in spite of any failures of links, nodes, and processes. Process resilience, reliable communication, distributed commit, checkpointing and recovery, agreement and consensus, failure detection, and self-stabilization are some of the mechanisms to provide fault-tolerance.
Security Distributed systems security involves various aspects of cryptography, secure channels, access control, key management – generation and distribution, authorization, and secure group management.
Scalability and modularity The algorithms, data (objects), and services must be as distributed as possible. Various techniques such as replication, caching and cache management, and asynchronous processing help to achieve scalability.

Algorithmic challenges in distributed computing

Designing useful execution models and frameworks The *interleaving* model and *partial order* model are two widely adopted models of distributed system executions. They have proved to be particularly useful for operational reasoning and the design of distributed algorithms.
Dynamic distributed graph algorithms and distributed routing algorithms The distributed system is modeled as a distributed graph, and the graph algorithms form the building blocks for a large number of higher level communication, data dissemination, object location, and object search functions.
Time and global state in a distributed system The processes in the system are spread across three-dimensional physical space. Another dimension, time, has to be superimposed uniformly across space. The challenges pertain to providing accurate *physical time*, and to providing a variant of time, called *logical time logical time can (i) capture the logic and inter-process dependencies within the distributed program, and also (ii) track the relative progress at each process.*
Synchronization/coordination mechanisms The processes must be allowed to execute concurrently, except when they need to synchronize to exchange information, i.e., communicate about shared data. The synchronization mechanisms can also be viewed as resource management and concurrency management mechanisms. Here are some examples of problems requiring synchronization:

• **Physical clock synchronization** Physical clocks ususally diverge in their values due to hardware limitations. Keeping them synchronized is a fundamental challenge to maintain common time.
• **Leader election** All the processes need to agree on which process will play the role of a distinguished process – called a leader process. A leader is necessary even for many distributed algorithms because there is often some asymmetry – as in initiating some action like a broadcast or collecting the state of the system, or in "regenerating" a token that gets "lost" in the system.
• **Mutual exclusion** This is clearly a synchronization problem because access to the critical resource(s) has to be coordinated.
• **Deadlock detection and resolution** Deadlock detection should be coordinated to avoid duplicate work, and deadlock resolution should be coordinated to avoid unnecessary aborts of processes.
Group communication, multicast, and ordered message delivery A group is a collection of processes that share a common context and collaborate on a common task within an application domain. Specific algorithms need to be designed to enable efficient group communication and group management
Distributed program design and verification tools Methodically designed and verifiably correct programs can greatly reduce the overhead of software design, debugging, and engineering. Designing mechanisms to achieve these design and verification goals is a challenge.
Debugging distributed programs Debugging sequential programs is hard; debugging distributed programs is that much harder because of the concurrency in actions and the ensuing uncertainty due to the large number of possible executions defined by the interleaved concurrent actions. Adequate debugging mechanisms and tools need to be designed to meet this challenge.

Applications of distributed computing

Mobile systems Mobile systems typically use wireless communication which is based on electromagnetic waves and utilizes a shared broadcast medium. There are two popular architectures for a mobile network. The first is the *base-station* approach, also known as the *cellular approach*, wherein a *cell* which is the geographical region within range of a static but powerful base transmission station is associated with that base station. All mobile processes in that cell communicate with the rest of the system via the base station. The second approach is the *ad-hoc network* approach where there is no base station (which essentially acted as a centralized node for its cell). All responsibility for communication is distributed among the mobile nodes, wherein mobile nodes have to participate in routing by forwarding packets of other pairs of communicating nodes.
Sensor networks A sensor is a processor with an electro-mechanical interface that is capable of sensing physical parameters, such as temperature, velocity, pressure, humidity, and chemicals. Sensor networks have a wide range of applications. Sensors may be mobile or static; sensors may communicate wirelessly, although they may also communicate across a wire when they are statically installed. Sensors may have to self-configure to form an ad-hoc network, which introduces a whole new set of challenges, such as position estimation and time estimation.
Ubiquitous or pervasive computing Ubiquitous systems are essentially distributed systems; recent advances in technology allow them to leverage wireless communication and sensor and actuator mechanisms. They can be self-organizing and network-centric, while also being resource constrained.
Peer-to-peer computing Peer-to-peer (P2P) computing represents computing over an application layer network wherein all interactions among the processors are at a "peer" level, without any hierarchy among the processors. Thus, all processors are equal and play a symmetric role in the computation. P2P networks are typically self-organizing, and may or may not have a regular structure to the network.
Distributed agents Agents are software processes or robots that can move around the system to do specific tasks for which they are specially programmed. The agents cooperate as in an ant

colony, but they can also have friendly competition, as in a free market economy. Challenges in distributed agent systems include coordination mechanisms among the agents, controlling the mobility of the agents, and their software design and interfaces.

Distributed data mining Data mining algorithms examine large amounts of data to detect patterns and trends in the data, to *mine* or extract useful information. In many situations, the data is necessarily distributed and cannot be collected in a single repository, as in banking applications where the data is private and sensitive, or in atmospheric weather prediction where the data sets are far too massive to collect and process at a single repository in real-time. In such cases, efficient distributed data mining algorithms are required.

Grid computing Many challenges in making grid computing a reality include: scheduling jobs in such a distributed environment, a framework for implementing quality of service and real-time guarantees, and, of course, security of individual machines as well as of jobs being executed in this setting.

Security in distributed systems The traditional challenges of security in a distributed setting include: confidentiality (ensuring that only authorized processes can access certain information), authentication (ensuring the source of received information and the identity of the sending process), and availability (maintaining allowed access to services despite malicious actions). The goal is to meet these challenges with efficient and scalable solutions.

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.

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

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 $\mathbf{e_i^x}$ denote the xth event at process $\mathbf{p_i}$.
- 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.
- The events at a process are linearly ordered by their order of occurrence.
- The execution of process pi produces a sequence of events $\mathbf{e_i^1}$, $\mathbf{e_i^2}$, ..., $\mathbf{e_i^x}$, $\mathbf{e_i^{x+1}}$, ... and is denoted by $\mathbf{H_i}$ where

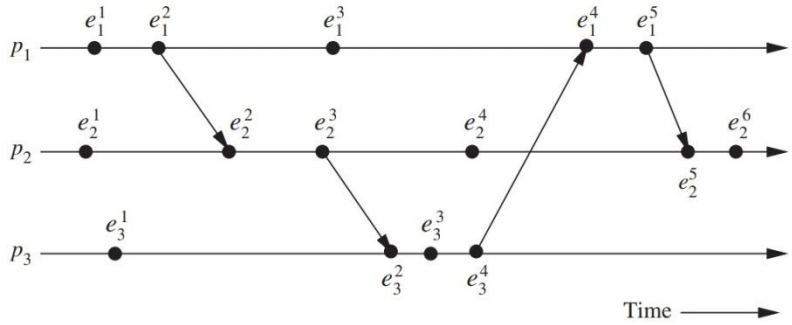
$$\mathbf{H_i = (h_i , \rightarrow i)}$$

$\mathbf{H_i}$ is the set of events produced by pi and binary relation $\rightarrow_i$ defines a linear order on these events.

SAVIOMANUEL

- 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.
- 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, for process p1, the second event is a message send event, the third event is an internal event, and the fourth event is a message receive event.



The space–time diagram of a distributed execution.

Causal Precedence Relation

The execution of a distributed application results in a set of distributed events produced by the processes.

Let $\mathbf{H=\cup_i h_i}$ 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.

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

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

Note that the relation → 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 transitively 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 the above figure, $e1^1 \rightarrow e3^3$ and $e3^3 \rightarrow e2^6$. The relation → 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 above figure, event $e2^6$ has the knowledge of all other events shown in the figure.

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

The three main types of communication models in distributed systems are:

- *FIFO (first-in, first-out):* each channel acts as a FIFO message queue.
- *Non-FIFO (N-FIFO):* a channel acts like a set in which a sender process adds messages and receiver removes messages in random order.
- *Causal Ordering (CO):* It follows Lamport's law i.e Lamport's "happens before" relation.

The relation between the three models is given by **CO ⊂ FIFO ⊂ Non-FIFO.**

A system that supports the causal ordering model satisfies the following property:

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

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 ⊂ FIFO ⊂ 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

The global state of a distributed system is a collection of the local states of its components, namely, the processes and the communication channels

The state of a process at any time 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 a 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.

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:

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

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 $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j,z_k}\}$ is a *consistent global state* iff

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

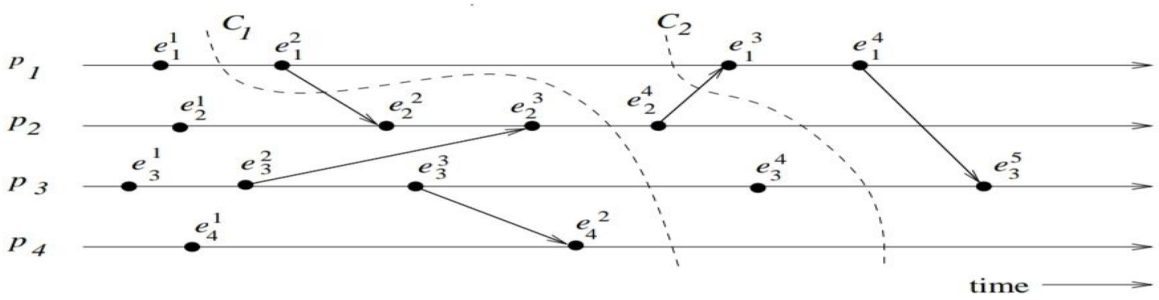
- That is, channel state $SC_{ij}^{y_i,z_k}$ and process state $LS_j^{z_k}$ must not include any message that process p_i sent after executing event $e_i^{x_i}$.

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.



Consistent cut: A consistent global state corresponds to a cut in which every message received in the PAST of the cut was sent in the PAST of that cut.
Inconsistent cut: A cut is inconsistent if a message crosses the cut from the FUTURE to the PAST.

- In a consistent cut, every message received in the PAST of the cut was sent in the PAST of that cut. (In Figure, cut C2 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, cut C1 is an inconsistent cut.)

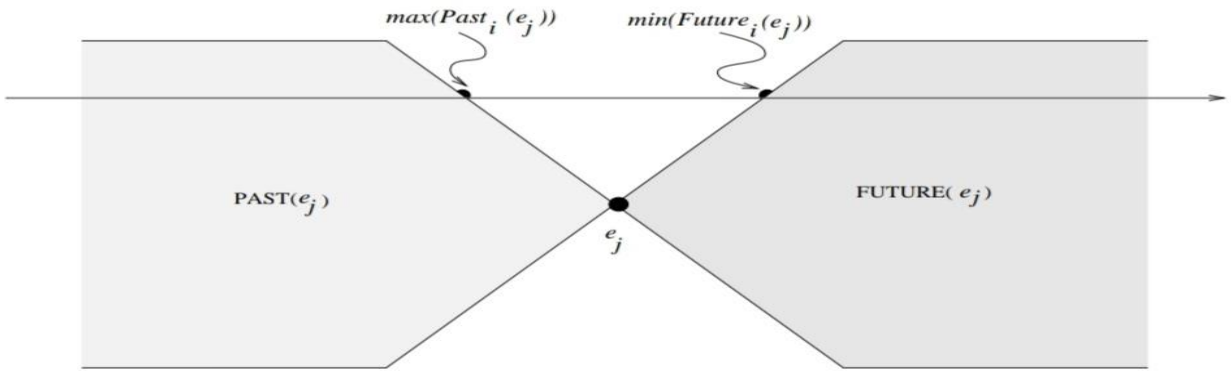
Past and Future Cones of an Event

Past Cone of an Event

- An event ej could have been affected only by all events e_i such that $e_i \rightarrow e_j$.
- In this situtaion, all the information available at ei could be made accessible at e_j .
- All such events e_i belong to the past of e_j .
- Let Past(e_j) denote all events in the past of e_j in a computation (H, →). Then,

  Past(e_j) = {e_i | ∀ e_i ∈ H, $e_i \rightarrow e_j$ }.
- Figure shows the past of an event 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 $Max\_Past(e_j) = \bigcup_{(\forall i)}\{max(Past_i(e_j))\}$.
- $Max\_Past(e_j)$ 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.

Future Cone of an Event

- 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:

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

Likewise we can define the following:

- 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 $\bigcup_{(\forall i)}\{min(Future_i(e_j))\}$, 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.

Models of Process Communications

There are two basic models of process communications

Synchronous: The sender process blocks until the message has been received by the receiver process. The sender process resumes executiononly after it learns that the receiver process has accepted the message. The sender and the receiver processes must synchronize to exchange a message.

Asynchronous: It is non- blocking communication where the sender and the receiver do not synchronize to exchange 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, thus causing a message burst.

Asynchronous communication achieves high degree of parallelism and non- determinism at the cost of implementation complexity with buffers. On the other hand, synchronization is simpler with low performance. The occurrence of deadlocks and frequent blocking of events prevents it from reaching higher performance levels.

****

SAVION MANUEL