UNIT I - INTRODUCTION

Introduction: Definition-Relation to Computer System Components – Motivation – Message -Passing Systems versus Shared Memory Systems – Primitives for Distributed Communication – Synchronous versus Asynchronous Executions – Design Issues and Challenges; A Model of Distributed Computations: A Distributed Program – A Model of Distributed Executions – Models of Communication Networks – Global State of a Distributed System.

Introduction:

1.1 Definition

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved.

A distributed system can be characterized as a collection of mostly autonomous processors communicating over a communication network and having the following features:

- **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.
- **No shared memory -** This is a key feature that requires message-passing for communication. This feature implies the absence of the common physical clock
- **Geographical separation -** The geographically wider apart that the processors are, the more representative is the system of a distributed system.
- **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.

1.2 Relation to Computer System Components

A typical distributed system is shown in Figure 1.1. Each computer has a memory-processing unit and the computers are connected by a communication network.

Figure 1.2 shows the relationships of the software components that run on each of the computers and use the local operating system and network protocol stack for functioning. 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.

Figure 1.1 A distributed system connects processors by a communication network.

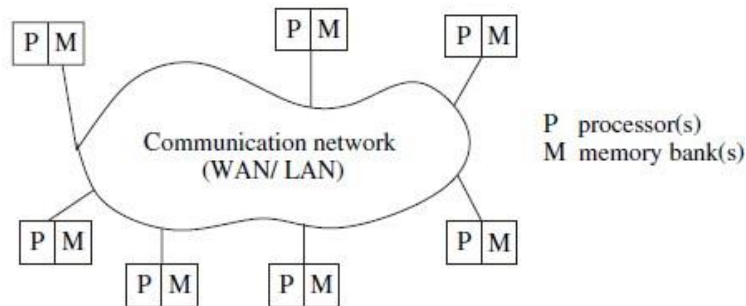
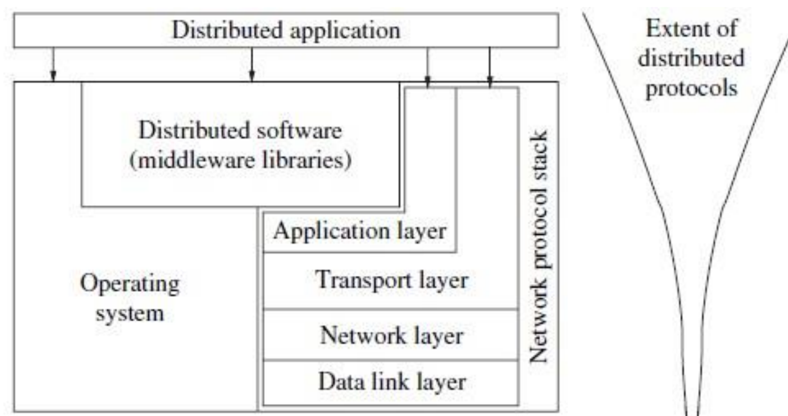


Figure 1.2 Interaction of the software components at each processor.



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.

Figure 1.2 schematically shows the interaction of this software with these system components at each processor.

Assume that the middleware layer does not contain the traditional application layer functions of the network protocol stack, such as http, mail, ftp, and telnet. Various primitive and

calls to functions defined in various libraries of the middleware layer are embedded in the user program code.

There exist several libraries to choose from to invoke primitives for the more common functions such as reliable and ordered multicasting of the middleware layer.

There are several standards such as Object Management Group's (OMG) common object request broker architecture (CORBA), and the remote procedure call (RPC) mechanism.

The RPC mechanism conceptually works like a local procedure call, with the difference that the procedure code may reside on a remote machine, and the RPC software sends a message across the network to invoke the remote procedure. It then awaits a reply, after which the procedure call completes from the perspective of the program that invoked it.

Currently deployed commercial versions of middleware often use CORBA, DCOM (distributed component object model), Java, and RMI (remote method invocation) technologies. The message-passing interface (MPI) developed in the research community is an example of an interface for various communication functions.

1.3 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. For example, distributed databases such as DB2 partition the data sets across several servers, in addition to replicating them at a few sites for rapid access as well as reliability
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.

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
5. **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.
6. **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.
7. **Modularity and incremental expandability:** Heterogeneous processors may be easily added into the system without affecting the performance, as long as those processors are running the same middleware algorithms. Similarly, existing processors may be easily replaced by other processors.

1.4 Message Passing Systems versus Shared Memory Systems

Shared memory systems are those in which there is a (common) shared address space throughout the system. Communication among processors takes place via shared data variables, and control variables for synchronization among the processors. Semaphores and monitors that were originally designed for shared memory uniprocessors and multiprocessors are examples of how synchronization can be achieved in shared memory systems.

All multicomputer (NUMA as well as message-passing) systems that do not have a shared address space provided by the underlying architecture and hardware necessarily communicate by message passing. Conceptually, programmers find it easier to program using shared memory than by message passing.

For this and several other reasons that we examine later, the abstraction called shared memory is sometimes provided to simulate a shared address space. For a distributed system, this abstraction is called distributed shared memory. Implementing this abstraction has a certain cost but it simplifies the task of the application programmer. There also exists a well-known folklore result that communication via message-passing can be simulated by communication via shared memory and vice-versa. Therefore, the two paradigms are equivalent.

Emulating message-passing on a shared memory system (MP →SM)

The shared address space can be partitioned into disjoint parts, one part being assigned to each processor. "Send" and "receive" operations can be implemented by writing to and reading from the destination/sender processor's address space, respectively.

Specifically, a separate location can be reserved as the mailbox for each ordered pair of processes. A Pi–Pj message-passing can be emulated by a write by Pi to the mailbox and then a read by Pj from the mailbox. In the simplest case, these mailboxes can be assumed to have unbounded size. The write and read operations need to be controlled using synchronization primitives to inform the receiver/sender after the data has been sent/received.

Emulating shared memory on a message-passing system (SM →MP)

This involves the use of "send" and "receive" operations for "write" and "read" operations. Each shared location can be modeled as a separate process; "write" to a shared location is emulated by sending an update message to the corresponding owner process; a "read" to a shared location is emulated by sending a query message to the owner process. As accessing another processor's memory requires send and receive operations, this emulation is expensive.

Thus, the latencies involved in read and write operations may be high even when using shared memory emulation because the read and write operations are implemented by using network-wide communication under the covers.

In a MIMD message-passing multicomputer system, each "processor" may be a tightly coupled multiprocessor system with shared memory. Within the multiprocessor system, the processors communicate via shared memory. Between two computers, the communication is by message passing. As message-passing systems are more common and more suited for wide-area distributed systems, we will consider message-passing systems more extensively than we consider shared memory systems.

1.5 Primitives for distributed communication

Blocking/non-blocking, synchronous/asynchronous primitives

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.

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.

The following are some definitions of blocking/non-blocking and synchronous/ asynchronous primitives.

- **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.

```
Send(X, destination, handle_k)                    // handle_k is a return parameter
...
...
Wait(handle_1, handle_2, ..., handle_k, ..., handle_m)        // Wait always blocks
```

Fig. A non-blocking send primitive.

The code for a non-blocking Send would look as shown in Figure. 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.

If at the time that Wait() is issued, the processing for the primitive (whether synchronous or asynchronous) has completed, the Wait returns immediately. The completion of the processing of the primitive is detectable by checking the value of *handlek*. If the processing of the primitive has not completed, the Wait blocks and waits for a signal to wake it up. When the processing for the primitive completes, the communication subsystem software sets the value of *handlek* and wakes up (signals) any process with a Wait call blocked on this *handlek* This is called posting the completion of the operation.

There are therefore four versions of the Send primitive – synchronous blocking, synchronous non-blocking, asynchronous blocking, and asynchronous non-blocking. For theReceive primitive, there are the blocking synchronous and non-blocking synchronous versions. These versions of the primitives are illustrated in Figure using a timing diagram. Here the timelines are shown for each process: (1) for the process execution, (2) for the user buffer from/to whichdata is sent/received, and (3) for the kernel/communication subsystem.

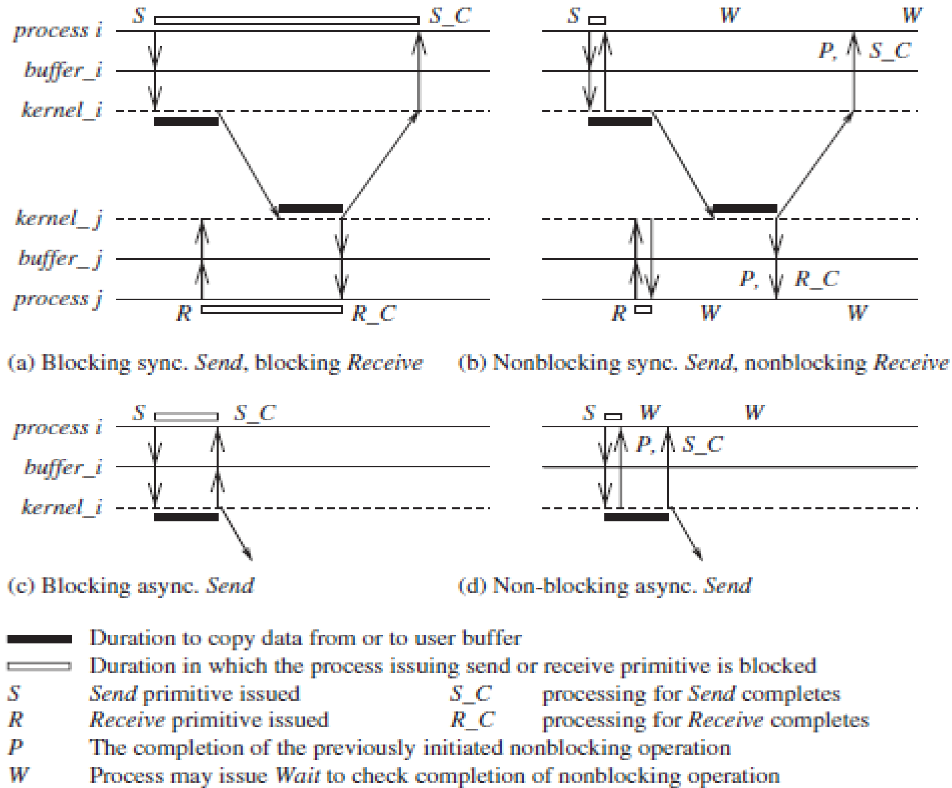


Figure: Blocking/ non-blocking and synchronous/asynchronous primitives.
Process Pi is sending and process Pj is receiving.
(a) Blocking synchronous Send and blockin (synchronous) Receive.
(b) Non-blocking synchronous Send and nonblocking (synchronous) Receive.
(c) Blocking asynchronous Send.
(d) Non-blocking asynchronous Send.

Processor synchrony

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.

Libraries and standards

There exists a wide range of primitives for message-passing. Many commercial software products (banking, payroll, etc., applications) use proprietary primitive libraries supplied with the software marketed by the vendors (e.g., the IBM CICS software which has a very widely installed customer base worldwide uses its own primitives).

The message-passing interface (MPI) library and the PVM (parallel virtual machine) library are used largely by the scientific community, but other alternative libraries exist.

Commercial software is often written using the remote procedure calls (RPC) mechanism in which procedures that potentially reside across the network are invoked transparently to the user, in the same manner that a local procedure is invoked.

1.6 Synchronous versus Asynchronous Executions

An asynchronous execution is an execution in which,

(i) there is no processor synchrony and there is no bound on the drift rate of processor clocks,

(ii) message delays (transmission + propagation times) are finite but unbounded,

(iii) there is no upper bound on the time taken by a process to execute a step.

An example asynchronous execution with four processes P0 to P3 is shown in Figure. The arrows denote the messages; the tail and head of an arrow mark the send and receive event for that message, denoted by a circle and vertical line, respectively. Non-communication events, also termed as internal events, are shown by shaded circles.

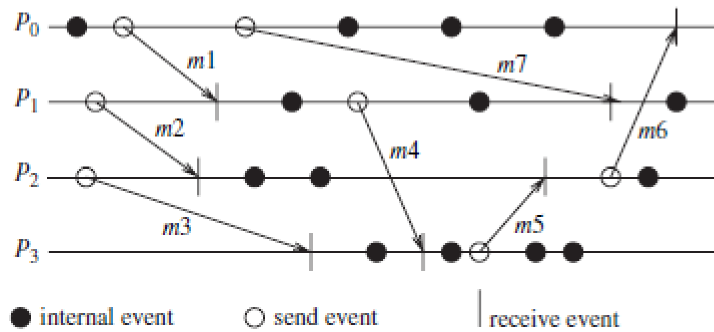


Figure: An example of an asynchronous execution in a message-passing system

A synchronous execution is an execution in which

(i) processors are synchronized and the clock drift rate between any two processors isbounded,

(ii) message delivery (transmission + delivery) times are such that they occur in onelogical step or round,

(iii) there is a known upper bound on the time taken by a process to execute a step.

An example of a synchronous execution with four processes P0 to P3 is shown in Figure. The arrows denote the messages.

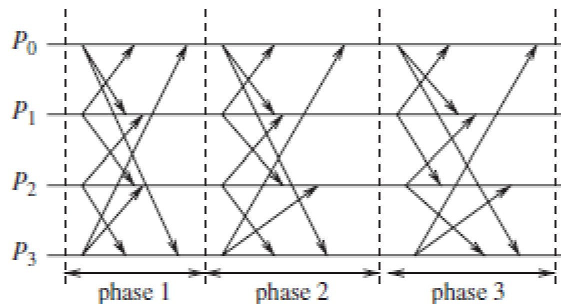


Figure: An example of a synchronous execution in a message-passing system

The synchronous execution is an abstraction that needs to be provided to the programs. When implementing this abstraction, observe that the fewer the steps or "synchronizations" of the processors, the lower the delays and costs. If processors are allowed to have an asynchronous execution for a period of time and then they synchronize, then the granularity of the synchrony is

coarse. This is really a virtually synchronous execution, and the abstraction is sometimes termed as virtual synchrony.

Ideally, many programs want the processes to execute a series of instructions in rounds (also termed as steps or phases) asynchronously, with the requirement that after each round/step/phase, all the processes should be synchronized and all messages sent should be delivered. This is the commonly understood notion of a synchronous execution. Within each round/phase/step, there may be a finite and bounded number of sequential sub-rounds (or subphases or sub-steps) that processes execute. Each sub-round is assumed to send at most onemessage per process; hence the message(s) sent will reach in a single message hop.

Emulating an asynchronous system by a synchronous system (A→S)

An asynchronous program (written for an asynchronous system) can be emulated on a synchronous system fairly trivially as the synchronous system is a special case of an asynchronous system – all communication finishes within the same round in which it is initiated.

Emulating a synchronous system by an asynchronous system (S →A)

A synchronous program (written for a synchronous system) can be emulated on an asynchronous system using a tool called synchronizer.

Emulations

Figure: Emulations among the principal system classes in a failure free system.

There are four broad classes of programs, as shown in Figure. Using the emulations shown, any class can be emulated by any other. If system A can be emulated by system B denoted A/B, and if a problem is not solvable in B, then it is also not solvable in A. Likewise, if a problem is

solvable in A, it is also solvable in B. Hence, in a sense, all four classes are equivalent in terms of "computability" – what can and cannot be computed – in failure-free systems.

1.7 Design issues and challenges

Distributed systems challenges from a system perspective

a. **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 cation (ROI), message-oriented communication versus stream-oriented communication.
b. **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.
c. **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.
d. **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.
e. **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.
f. **Consistency and replication**: To avoid bottlenecks, to provide fast access to data, and to provide scalability, replication of data objects is highly desirable.
g. **Fault tolerance:** Fault tolerance requires maintaining correct and efficient operation in spite of any failures of links, nodes, and processes.
h. **Security:** Distributed systems security involves various aspects of cryptography, secure channels, access control, key management – generation and distribution, authorization, and secure group management.
i. **Applications Programming Interface (API) and transparency**: The API for communication and other specialized services is important for the ease of use and wider adoption of the distributed systems services by non-technical users.

   Transparency deals with hiding the implementation policies from the user, and can be classified as follows.

a. Access Transparency hides differences in data representation on different systems and provides uniform operations to access system resources.
b. Location transparency makes the locations of resources transparent to the users.
c. Migration transparency allows relocating resources without changing names.
d. The ability to relocate the resources as they are being accessed is relocation transparency.
e. Replication transparency does not let the user become aware of any replication.
f. Concurrency transparency deals with masking the concurrent use of shared resources for the user.
g. Failure transparency refers to the system being reliable and fault-tolerant.

j. **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

a. 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.

b. 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.

c. 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.

d. 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. Synchronization is essential for the distributed processes to overcome the limited

observation of the system state from the viewpoint of any one process. Here are some examples of problems requiring synchronization. They are Physical clock synchronization, Leader election, Mutual exclusion, Deadlock detection and resolution, Termination detection and Garbage collection.

e. 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 wherein processes can join and leave groups dynamically, or even fail.

f. Monitoring distributed events and predicates

Predicates defined on program variables that are local to different processes are used for specifying conditions on the global system state, and are useful for applications such as debugging, sensing the environment, and in industrial process control.

g. 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

h. 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.

i. Data replication, consistency models, and caching

Fast access to data and other resources requires them to be replicated in the distributed system. Managing such replicas in the face of updates introduces the problems of ensuring consistency among the replicas and cached copies.

j. World Wide Web design – caching, searching, scheduling

The Web is an example of a widespread distributed system with a direct interface to the end user, wherein the operations are predominantly read-intensive on most objects.

k. Distributed shared memory abstraction

A shared memory abstraction simplifies the task of the programmer because he or she has to deal only with read and write operations, and no message communication primitives. However, under the covers in the middleware layer, the abstraction of a shared address

space has to be implemented by using message-passing. Hence, in terms of overheads, the shared memory abstraction is not less expensive.

l. Reliable and fault-tolerant distributed systems

A reliable and fault-tolerant environment has multiple requirements and aspects, and these can be addressed using various strategies. They are Consensus algorithms, Replication and replica management, Voting and quorum systems, Distributed databases and distributed commit, Self-stabilizing systems, Checkpointing and recovery algorithms, Failure detectors.

m. Load balancing

The goal of load balancing is to gain higher throughput, and reduce the userperceived latency. The following are some forms of load balancing: Data migration, Computation migration and Distributed scheduling.

n. Real-time scheduling

Real-time scheduling is important for mission-critical applications, to accomplish the task execution on schedule. The problem becomes more challenging in a distributed system where a global view of the system state is absent. On-line or dynamic changes to the schedule are also harder to make without a global view of the state.

o. Performance

Although high throughput is not the primary goal of using a distributed system, achieving good performance is important. The following are some example issues arise in determining the performance: Metrics and Measurement methods/tools

Applications of distributed computing and newer challenges

- **Mobile systems**

  Mobile systems typically use wireless communication which is based on electromagnetic waves and utilizes a shared broadcast medium. Hence, the characteristicsof communication are different; many issues such as range of transmission and power of transmission come into play, besides various engineering issues such as battery power conservation, interfacing with the wired Internet, signal processing and interference

- **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. Recent developments in cost-effective hardware technology have made it possible to deploy very large (of the order of 106 or higher) low-cost sensors.

- **Ubiquitous or pervasive computing**

  Ubiquitous systems represent a class of computing where the processors embedded in and seamlessly pervading through the environment perform application functions in the background, much like in sci-fi movies. The intelligent home, and the smart workplace are some example of ubiquitous environments currently under intense research and development.

- **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.

- **Publish-subscribe, content distribution, and multimedia**

  In a dynamic environment where the information constantly fluctuates (varying stock prices is a typical example), there needs to be:

  (i) an efficient mechanism for distributing this information (publish),

  (ii) an efficient mechanism to allow end users to indicate interest in receiving specific kinds of information (subscribe)

  (iii) an efficient mechanism for aggregating large volumes of published information and filtering it as per the user's subscription filter

- **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 name "agent" derives from the fact that the agents do work on behalf of some broader objective.

- **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. A traditional example is: examining the purchasing patterns of customers in order to profile the customers and enhance the efficacy of directed marketing schemes.

- **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 sendingprocess), and availability (maintaining allowed access to services despite malicious actions). The goal is to meet these challenges with efficient and scalable solutions.

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 p_i to process p_j and let m_{ij} denote a message sent by p_i 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 atwhich 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 $e_i^1, e_i^2, \ldots, e_i^x, \ldots, e_i^{x+1}, \ldots$ 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

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

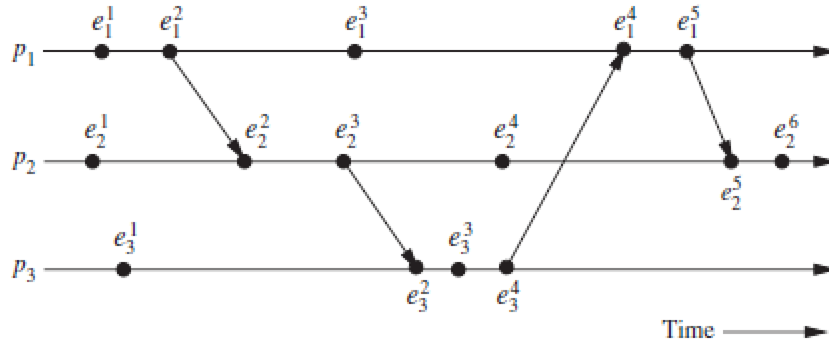


Figure: The space–time diagram of a distributed execution.

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 weassume 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 p1, the second event is a messagesend 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 occurred at the same instant in physical time.

1.10 Models of communication networks

There are several models of the service provided by communication networks, namely, FIFO (first-in, first-out), 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.

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 $rec(m_{ij}) \longrightarrow rec(m_{kj})$.

That is, 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. Furthermore, note that CO ⊂ FIFO ⊂ Non-

FIFO. Causal ordering model is useful in developing distributed algorithms.

Generally, it considerably simplifies the design of distributed algorithms because it provides a built-in synchronization. For example, in replicated database systems, it is important that every process responsible for updating a replica receives the updates in the same order to maintain database consistency

1.11 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 localcontext of the distributed application. The state of a channel is given by the set of messages in transit in the channel.

Global state

The global state of a distributed system is a collection of the local states of the processes and the channels. Notationally, the 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 snapshot 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 there was a global system clock that could be instantaneously read by the processes. However, both are impossible.

However, it turns out that even if the state of all the components in a distributed system has not been 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 an effect should not be present without its cause. A message cannot be received if it was not sent; that is, the state should not violate causality. 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 it satisfies the following condition:

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

That is, channel state $SC_{ik}^{y_i,z_k}$ and process state $LS_k^{z_k}$ must not include any message that process *pi*

sent after executing event e_i^{xi}.

In the distributed execution of Figure, a global state GS1 consisting of local states $\{LS^1, LS^3, LS^3, LS^2\}$ is inconsistent because the state of p2 has recorded the receipt of message m12, however, the state of p1 has not recorded its send. On the contrary, a global state GS2 consisting of local states $\{LS^1, LS^3, LS^3, LS^2\}$ is consistent; all the channels are empty except C21 that contains message

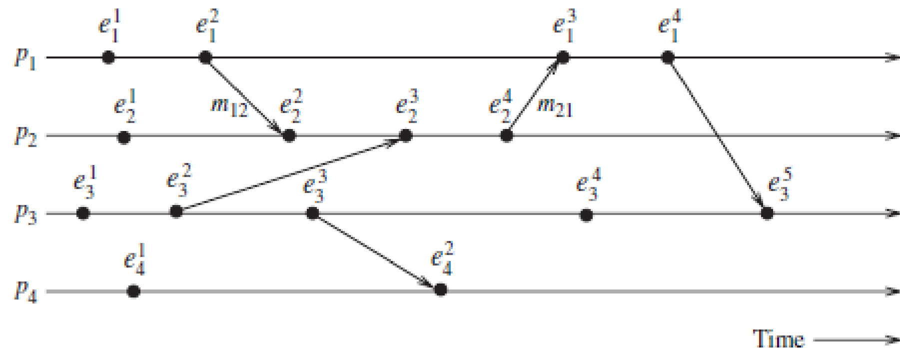


Figure: The space–time diagram of a distributed execution.

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

$$\forall i, \forall j \ : 1 \leq i, j \ \leq n \ :: SC_{ij}^{y_i,z_j} = \phi.$$

Thus, all channels are recorded as empty in a transitless global state. A global state isstrongly consistent iff it is transitless as well as consistent. Note that in Figure, the global state
consisting of local states $\{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is strongly consistent.

Recording the global state of a distributed system is an important paradigm when one is interested in analyzing, monitoring, testing, or verifying properties of distributed applications, systems, and algorithms. Design of efficient methods for recording the global state of a distributedsystem is an important problem.