DISTRIBUTED SYSTEMS

CHAPTER – 1

INTRODUCTION OF DISTRIBUTED SYSTEMS

1.1 Definition

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. In other way a large amount of problem that can be solved using a group of computer systems which are communicating over a network.

These computer systems are having the following common features:

a. No Common Physical Clock

Each system having their own system clock, so the processor considering the execution time with its own clock only. Therefore, the communication among the processors in the group of systems are asynchronous way.

b. No Shared Memory

Each computer system in the group are having its own memory to solve the part of the problem. Note that a distributed system provides the abstraction of a common address space via the distributed shared memory abstraction.

c. Geographical Separation

The communication networks of the distributed systems is not necessary on a wide-area network (WAN), because of the Network Of Workstation (NOW) or Cluster Of Workstation (COW) configuration for the connection of processor on a LAN is used in a small distributed system. Google Search Engine is based on the NOW architecture, because of NOW is “low-cost” and “high-speed”.

d. Autonomy and Heterogeneity

The processors are “loosely coupled” in that they have different speeds and each can be running a different OS. They are cooperate with one another by offering Services (or) Solving a problem jointly.

1.2 Relation to Computer System Components

Each distributed systems has a “memory-processing” unit and the systems are connected by a communication network.

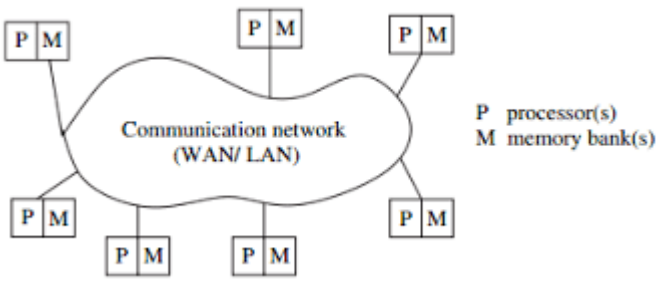


Figure 1.1 A Distributed System connects Processors by a Communication Network

The distributed software is also termed as “middleware” and the relationship 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 system uses a layered architecture to reduce the complexity of system design. The middleware is the distributed software that 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. There exist several libraries to choose from to invoke primitives for the more common functions, such as “reliable” and “ordered multi-casting” of the middleware layer.

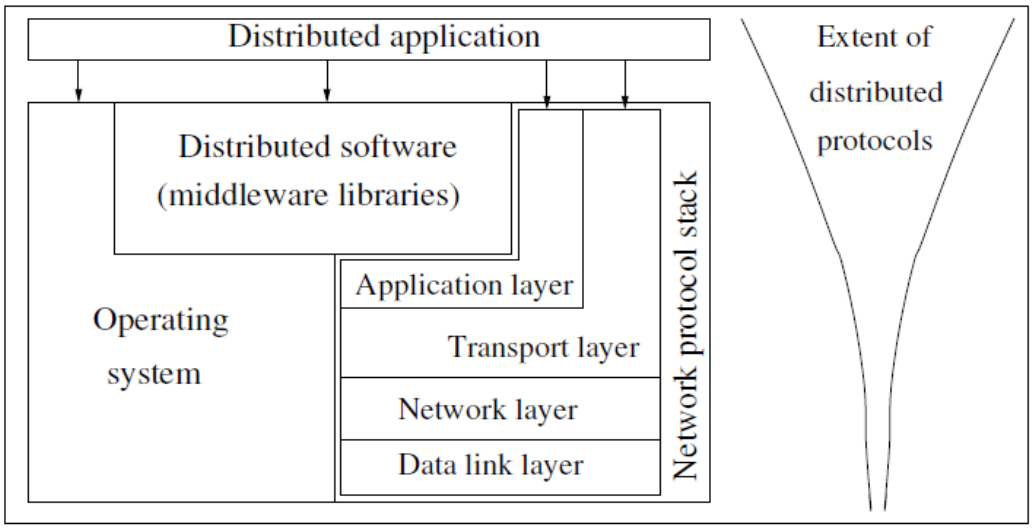


Figure 1.2: Interaction of the software components at each processor

There are several standard libraries:

a. Message Passing Interface (Remote Procedure Call (RPC))
The RPC mechanism conceptually works like a local procedure call, the procedure code resided on a remote machine. The RPC software sends a message across the network to invoke the remote procedure code. Then wait for a reply, after which the procedure call completes from the perspective of the program that invoked it.

b. Object Passing Interface (Java and RMI, CORBA and DCOM)
Currently deployed commercial versions of middleware often use CORBA (Common Object Resource Broker Architecture), DCOM (Distributed Component Object Model), Java, and RMI (Remote Method Invocation) technologies.

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. 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.

Advances in the design of resource-constrained mobile devices as well as in the wireless technology with which these devices communicate have given further impetus to the importance of distributed protocols and middleware.

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:*** The resource should be accessible at all times;
- ***Integrity: T***he 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:*** The ability to recover from system failures, where such failures may be defined to occur in one of many failure models.

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. Such a configuration provides a better performance/cost ratio than using special parallel machines. This is particularly true of the NOW configuration.
In addition to meeting the above requirements, a distributed system also offers the following advantages:

6. Scalability As the processors are usually connected by a wide-area net- work, 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 Relation to parallel systems

The distributed system can be identified to meet the above characteristics. Then how can one classify the system is parallel multiprocessor system or distributed system. Therefore we are going to examine both (Multi-Processsor / Multi-Computer) systems taxonomies.

1.4.1 Characteristics of Parallel Systems

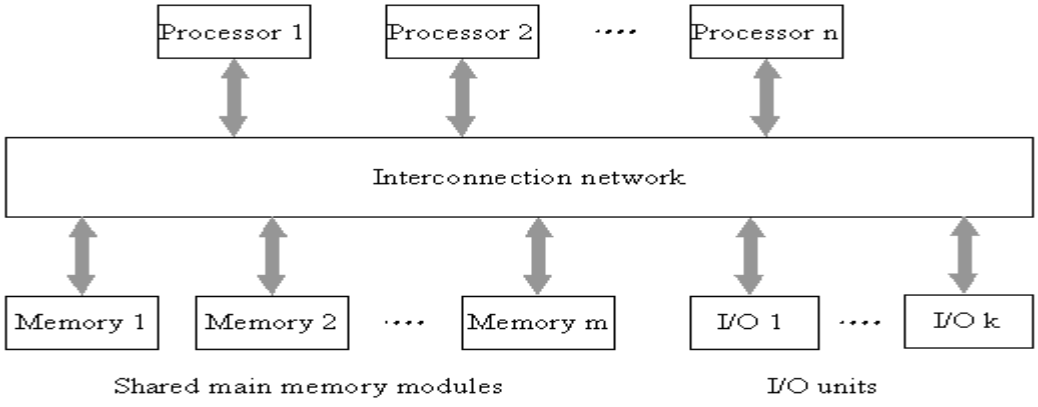
A parallel system may be broadly classified as belonging to one of three types:

1. Multi-processor System (UMA Architecture) – No Common Clock
2. Multi-computer System (NUMA Architecture) – No Common Clock
3. Array Processors – Having Common Clock

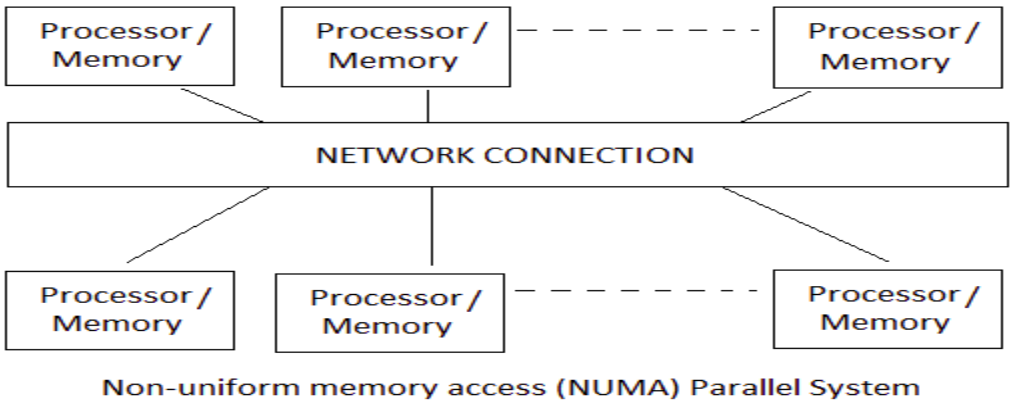
1. A multiprocessor system is a parallel system in which the multiple processors have direct access to shared memory which forms a common address space. Such processors usually do not have a common clock.

A multiprocessor system usually corresponds to a ***Uniform Memory Access (UMA) architecture*** in which the access latency, i.e., waiting time, to complete an access to any memory location from any processor is the same. The processors are in very close physical proximity and are connected by an interconnection network. Inter-process communication across processors is traditionally through read and write operations on the shared memory, although the use of message-passing primitives such as those provided by the MPI, is also possible (using emulation on the shared memory).

All the processors usually run the same operating system, and both the hardware and software are very tightly coupled. The processors are usually of the same type, and are housed within the same box/container with a shared memory.



(a)



(b)

Figure 1.3: Two standard architectures for parallel systems (a)UMA Multiprocessor system
(b) NUMA Multiprocessor system

In both architecture, the processors may locally cache data from memory.

The interconnection network to access the memory may be a bus, although for greater efficiency, it is usually a multistage switch with a symmetric and regular design.

We are going to study about two popular interconnection networks:

1. Omega Network 2. Butterfly Network

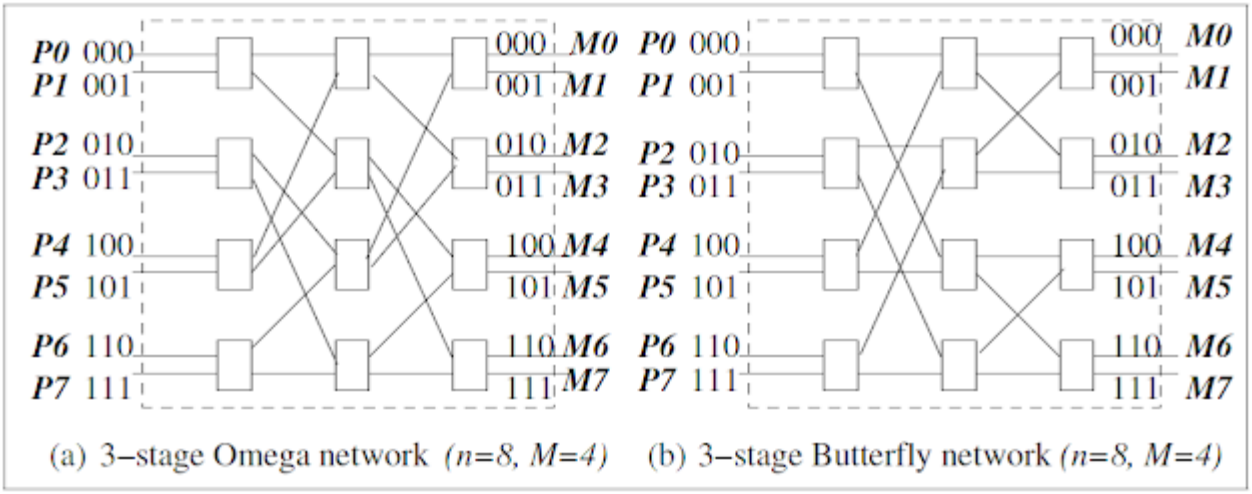


Figure 1.4 Interconnection networks for shared memory multiprocessor systems
(a) Omega Network for n=8 processors p0 to p7 and memory banks M0 to M7
(b) Butterfly Network for n=8 processors p0 to p7 and Memory Banks M0 to M7

Each network is a multi-stage network formed of 2x2 switching elements, and each switch allows data on either of the two input wires to be switched to the upper or the lower output wire.

So if the data from both the input wires is to be routed to the same output wire in a single step, there is a collision. Various techniques such as buffering or more elaborate interconnection designs can address collisions.

Each 2 × 2 switch is represented as a rectangle in the figure. Further- more, a n-input and n-output network uses log n stages and log n bits for addressing. Routing in the 2 × 2 switch at stage k uses only the kth bit, and hence can be done at clock speed in hardware. The multi-stage networks can be constructed recursively, and the interconnection pattern between any two stages can be expressed using an iterative or a recursive generating function.

1. Omega Network

The Omega network which connects n processors to n memory units has n/2log 2 n switching elements of size 2 × 2 arranged in log 2 n stages. Between each pair of adjacent stages of the

Omega network, a link exists between output i of a stage and the input j to the next stage according to the following perfect shuffle pattern which is a left-rotation operation on the binary representation of i to get j.

The iterative generation function is as follows:

$$j = \begin{cases} 2i, & \text{for } 0 \le i \le n/2 - 1, \\ 2i + 1 - n & \text{for } n/2 \le i \le n - 1. \end{cases}$$

Consider any stage of switches. Informally, the upper (lower) input lines for each switch come in sequential order from the upper (lower) half of the switches in the earlier stage.

With respect to the Omega network in Figure 1.4(a), n = 8. Hence, for any stage, for the outputs i, where $0 \le i \le 3$, the output i is connected to input 2i of the next stage. For $4 \le i \le 7$, the output i of any stage is connected to input $2i + 1 - n$ of the next stage.

Omega Routing Function

The routing function from input line i to output line j considers only j and the stage number s, where s ∈ #0# log 2 n − 1#. In a stage s switch, if the s + 1th MSB (most significant bit) of j is 0, the data is routed to the upper output wire, otherwise it is routed to the lower output wire.

2. Butterfly Network

The generation of the interconnection pattern between a pair of adjacent stages depends not only on “n” but also on the stage number “s”.

The recursive function

Let there be M = n/2 switches per stage, and let a switch be denoted by the tuple (x, s), where $x \in [0, M - 1]$ and stage $s \in [0, \log_2 n - 1]$.
The two outgoing edges from any switch (x, s) are as follows.

There is an edge from switch (x, s) to switch (y, s + 1) if (i) x = y or (ii) x XOR y has exactly one 1 bit, which is in the $(s + 1)^{th}$ MSB. For stage s, apply the rule above for $M/2^s$ switches.

Whether the two incoming connections go to the upper or the lower input port is not important because of the routing function, given below.

For Example:

Consider the Butterfly network in Figure 1.4(b), n = 8 and M = 4. There are three stages, s = 0, 1, 2, and the interconnection pattern is defined between s = 0 and s = 1 and between s = 1 and s = 2.

The switch number x varies from 0 to 3 in each stage, i.e., x is a 2-bit string.

Consider the first stage interconnection (s = 0) of a butterfly of size M, and hence having $\log_2 2M$ stages.

For stage s = 0, as per rule (i), the first output line from switch 00 goes to the input line of switch 00 of stage s = 1.

As per rule (ii), the second output line of switch 00 goes to input line of switch 10 of stage s = 1.

Similarly, x = 01 has one output line go to an input line of switch 11 in stage s = 1. The other connections in this stage can be determined similarly.

For stage s = 1 connecting to stage s = 2, we apply the rules considering only M/2 1 = M/2 switches, i.e., we build two butterflies of size M/2 – the “upper half” and the “lower half” switches.

The recursion terminates for M/2 s = 1, when there is a single switch.

Butterfly Routing Function

In a stage “s” switch, if the $s + 1^{th}$ MSB of j is 0, the data is routed to the upper output wire, otherwise it is routed to the lower output wire.

Conclusion

For the Butterfly and the Omega networks,

the paths from the different inputs to any one output form a spanning tree. This implies that collisions will occur when data is destined to the same output line.

However, the advantage is that data can be combined at the switches if the application semantics (e.g., summation of numbers) are known.

2. A multicomputer parallel system is a parallel system in which the multiple processors do not have direct access to shared memory. The memory of the multiple processors may or may not form a common address space. Such computers usually do not have a common clock.

The processors are in close physical proximity and are usually very tightly coupled (homogenous hardware and software), and connected by an interconnection network. The processors communicate either via a common address space or via message-passing. A multicomputer system that has a common address space usually corresponds to a non-uniform memory access (NUMA) architecture in which the latency to access various shared memory locations from the different processors varies.

Examples of parallel multicomputers are: the NYU Ultracomputer and the Sequent shared memory machines, the CM* Connection machine and processors configured in regular and symmetrical topologies such as an array or mesh, ring, torus, cube, and hypercube (message-passing machines).

The regular and symmetrical topologies have interesting mathematical properties that enable very easy routing and provide many rich features such as alternate routing.

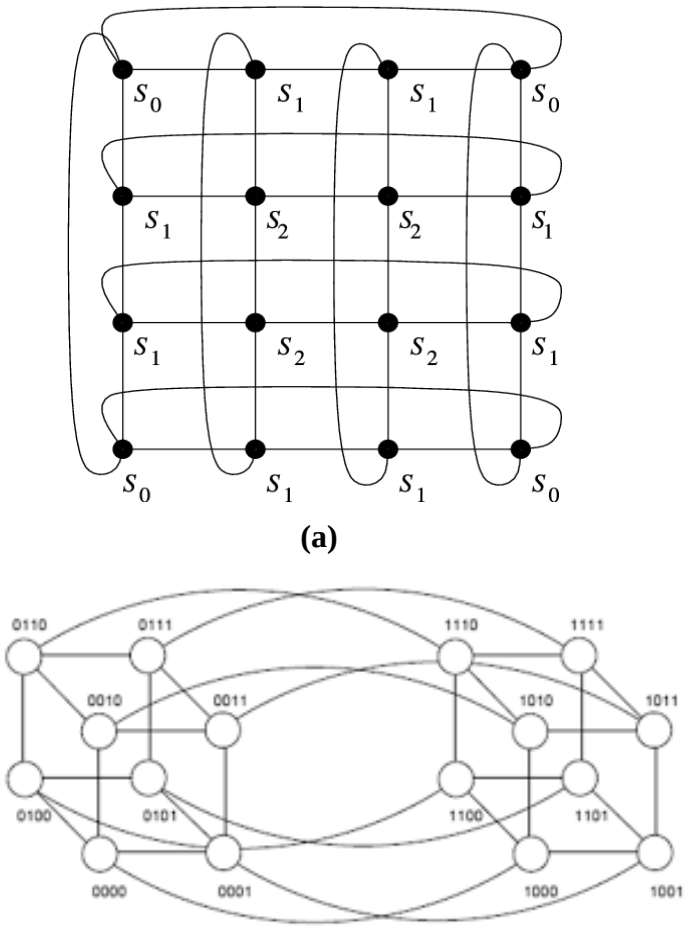


Figure 1.5: Some popular topologies for multicomputer shared-memory machines. (a) Wrap-around 2D mesh (torus) (b) Hypercube of dimension 4.

(a) Wrap-around 4 x 4 Mesh

For a k × k mesh which will contain k^2 processors, the maximum path length between any two processors is 2(k/2 − 1). Routing can be done along the Manhattan grid.

(b) Hypercube of dimension 4

A k-dimensional hypercube has 2 k processor-and-memory units. Each such unit is a node in the hypercube, and has a unique k-bit label. Each of the k dimensions is associated with a bit position in the label. The labels of any two adjacent nodes are identical except for the bit position corresponding to the dimension in which the two nodes differ. Thus, the processors are labeled such that the shortest path between any two processors is the Hamming distance (defined as the number of bit positions in which the two equal sized bit strings differ) between the processor labels. This is clearly bounded by k.

For example:

Nodes 0101 and 1100 have a Hamming distance of 2. The shortest path between them has length 2.

Routing in the hypercube is done hop-by-hop. At any hop, the message can be sent along any dimension corresponding to the bit position in which the current node's address and the destination address differ.

The 4D hypercube shown in the figure is formed by connecting the corresponding edges of two 3D hypercubes (corresponding to the left and right "cubes" in the figure) along the fourth dimension.

The labels of the 4D hypercube are formed by prepending a "0" to the labels of the left 3D hypercube and prepending a "1" to the labels of the right 3D hypercube. This can be extended to construct hypercubes of higher dimensions.

Conclusion: Observe that there are multiple routes between any pair of nodes, which provides fault- tolerance as well as a congestion control mechanism. The hypercube and its variant topologies have very interesting mathematical properties with implications for routing and fault-tolerance.

3. Array processors belong to a class of parallel computers that are physically co-located, are very tightly coupled, and have a common system clock (but may not share memory and communicate by passing data using messages).

Array processors and systolic arrays that perform tightly synchronized processing and data exchange in lock-step for applications such as DSP and Image Processing belong to this category. These applications usually involve a large number of iterations on the data.

The primary and most efficacious use of parallel systems is for obtaining a higher throughput by dividing the computational workload among the processors. The tasks that are most amenable to higher speedups on parallel systems are those that can be partitioned into sub tasks very nicely, involving much number-crunching and relatively little communication for synchronization. Once

the task has been decomposed, the processors perform large vector, array, and matrix computations that are common in scientific applications.
Early 1990s, they have not proved to be economically viable for two related reasons. First, the overall market for the applications that can potentially attain high speedups is relatively small. Second, due to economy of scale and the high processing power offered by relatively inexpensive off-the-shelf networked PCs, specialized parallel machines are not cost-effective to manufacture. They additionally require special compiler and other system support for maximum throughput.

1.4.2 Flynn's Taxonomy

Flynn identified four processing modes, based on whether the processors execute the same or different instruction streams at the same time, and whether or not the processors processed the same (identical) data at the same time. It is instructive to examine this classification to understand the range of options used for configuring systems:

Single instruction stream, single data stream (SISD)

This mode corresponds to the conventional processing in the von Neumann paradigm with a single CPU, and a single memory unit connected by a system bus.

Single instruction stream, multiple data stream (SIMD)

This mode corresponds to the processing by multiple homogenous processors which execute in lock-step on different data items. Applications that involve operations on large arrays and matrices, such as scientific applications, can best exploit systems that provide the SIMD mode of operation because the data sets can be partitioned easily.

Several of the earliest parallel computers, such as Illiac-IV, MPP, CM2, and MasPar MP-1 were SIMD machines. Vector processors, array processors' and systolic arrays also belong to the SIMD class of processing. Recent SIMD architectures include co-processing units such as the MMX units in Intel processors (e.g., Pentium with the streaming SIMD extensions (SSE) options) and DSP chips such as the Sharc.

Multiple instruction stream, single data stream (MISD)

This mode corresponds to the execution of different operations in parallel on the same data. This is a specialized mode of operation with limited but niche applications, e.g., visualization.

Multiple instruction stream, multiple data stream (MIMD)

In this mode, the various processors execute different code on different data. This is the mode of operation in distributed systems as well as in the vast majority of parallel systems. There is no common clock among the system processors. Sun Ultra servers, multicomputer PCs, and IBM SP machines are examples of machines that execute in MIMD mode.

MIMD architectures are most general and allow much flexibility in partitioning code and data to be processed among the processors. MIMD architectures also include the classically understood mode of execution in distributed systems.

1.4.3 Coupling, Parallelism, Concurrency, and Granularity

Coupling

The degree of coupling among a set of modules, whether hardware or software, is measured in terms of the inter dependency and binding and/or homogeneity among the modules. When the degree of coupling is high (low), the mod- ules are said to be tightly (loosely) coupled. SIMD and MISD architectures generally tend to be tightly coupled because of the common clocking of the shared instruction stream or the shared data stream.

Here we briefly examine various MIMD architectures in terms of coupling:

• Tightly coupled multiprocessors (with UMA shared memory). These may be either switch-based (e.g., NYU Ultracomputer, RP3) or bus-based (e.g., Sequent, Encore).

• Tightly coupled multiprocessors (with NUMA shared memory or that communicate by message passing). Examples are the SGI Origin 2000 and the Sun Ultra HPC servers (that communicate via NUMA shared memory), and the hypercube and the torus (that communicate by message passing).

• Loosely coupled multicomputers (without shared memory) physically co-located. These may be bus-based (e.g., NOW connected by a LAN or Myrinet card) or using a more general communication network, and the processors may be heterogenous. In such systems, processors neither share memory nor have a common clock, and hence may be classified as distributed systems – however, the processors are very close to one another, which is characteristic of a parallel system. As the communication latency may be significantly lower than in wide-area distributed systems, the solution approaches to various problems may be different for such systems than for wide-area distributed systems.

• Loosely coupled multicomputers (without shared memory and without common clock) that are physically remote. These correspond to the conventional notion of distributed systems.

Parallelism or speedup of a program on a specific system
This is a measure of the relative speedup of a specific program, on a given machine. The speedup depends on the number of processors and the mapping of the code to the processors. It is expressed as the ratio of the time T#1# with a single processor, to the time T#n# with n processors.

Parallelism within a parallel/distributed program
This is an aggregate measure of the percentage of time that all the processors are executing CPU instructions productively, as opposed to waiting for communication (either via shared memory or message-passing) operations to complete. The term is traditionally used to characterize parallel programs. If the aggregate measure is a function of only the code, then the parallelism is independent of the architecture. Otherwise, this definition degenerates to the definition of parallelism in the previous section.

Concurrency of a program
This is a broader term that means roughly the same as parallelism of a program, but is used in the context of distributed programs. The parallelism/concurrency in a parallel/distributed program can be measured by the ratio of the number of local (non-communication and non-shared memory

access) operations to the total number of operations, including the communication or shared memory access operations.

Granularity of a program

The ratio of the amount of computation to the amount of communication within the parallel/distributed program is termed as granularity. If the degree of parallelism is coarse-grained (fine-grained), there are relatively many more (fewer) productive CPU instruction executions, compared to the number of times the processors communicate either via shared memory or message- passing and wait to get synchronized with the other processors. Programs with fine-rained parallelism are best suited for tightly coupled systems. These typically include SIMD and MISD architectures, tightly coupled MIMD multiprocessors (that have shared memory), and loosely coupled multicomputers (without shared memory) that are physically co-located. If programs with fine-grained parallelism were run over loosely coupled multiprocessors that are physically remote, the latency delays for the frequent communication over the WAN would significantly degrade the overall throughput.

Various classes of multiprocessor/multicomputer operating systems:

• The operating system running on loosely coupled processors (i.e., heterogenous and/or geographically distant processors), which are themselves running loosely coupled software (i.e., software that is heterogenous), is classified as a network operating system. In this case, the application cannot run any significant distributed function that is not provided by the application layer of the network protocol stacks on the various processors.

• The operating system running on loosely coupled processors, which are running tightly coupled software (i.e., the middleware software on the processors is homogenous), is classified as a distributed operating system.

• The operating system running on tightly coupled processors, which are themselves running tightly coupled software, is classified as a multiprocessor operating system. Such a parallel system can run sophisticated algorithms contained in the tightly coupled software.

1.5 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.

For example:
Semaphores and monitors that were originally designed for shared memory uni-processors 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.

Programmers find it easier to program using shared memory than by message passing, this abstraction called shared memory. In a distributed system, this abstraction is called distributed shared memory.

The two well known paradigms "***Communication via Message-Passing***" and "***Communication via Shared Memory***" are equivalent.

1.5.1 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.

A separate location can be reserved as the mailbox for each ordered pair of processes. A P i –P j message-passing can be emulated by a write by P i to the mailbox and then a read by P j from the mailbox.

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.

1.5.2 Emulating Shared Memory on a Message-Passing System (SM → MP)

Each shared location can be modeled as a separate process;

A "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.

Emulating shared memory might seem to be more attractive from a programmer's perspective, it must be remembered that in a distributed system, it is only an abstraction.

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.

An application can of course use a combination of shared memory and message-passing.

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.

Conclusion:

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.

1.6 Primitives for Distributed Communication

In distributed communication systems, the primitives are used to Send and Receive the data.

(a) Message send denoted as **Send().**
It has at least two parameters:

1. Destination
2. Buffer in the User Space (containing data to be send)

Send() primitive used two ways of sending data.

1. Buffered Option: Copy the data from User Buffer to the Kernel Buffer. Later the data copied from Kernel Buffer on to the Network.
2. UnBuffered Option: Copy the data from User Buffer onto the Network.

(b) Message receive denoted as **Receive()**
It has at least two parameters:

1. Source from which the data is to be received (this could be a wildcard).
2. Buffer in User Space (to be Load the received data)

Receive() primitive used only one way of receiving data.

1. Buffered Option: The data arrived to the User Space, and needs a storage area in the Kernel.

The communication primitives are used in different ways on the following definitions.

a. Blocking and Non-Blocking
b. Synchronous and Asynchronous
c. Processors Synchrony
d. Libraries and Standards

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. Presumably after issuing the primitive in non-blocking mode, the process has done whatever actions it could and now needs to know the status of completion of the call, therefore using a blocking **Wait()** call is usual programming practice.

Send(X, destination, $handle_K$) // $handle_K$ is a return parameter
. . .
. . .

Wait($handle_1$, $handle_2$ … $handle_k$ … $handle_m$) // Wait always blocks

Posting the completion of the Operation

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.

Observation:

There are 4 versions of the Send primitives:

1. Synchronous Blocking
2. Synchronous Non-Blocking
3. Asynchronous Blocking
4. Asynchronous Non-Blocking

There are 2 versions of the Receive primitives:
1. Synchronous Blocking
2. Synchronous Non-Blocking

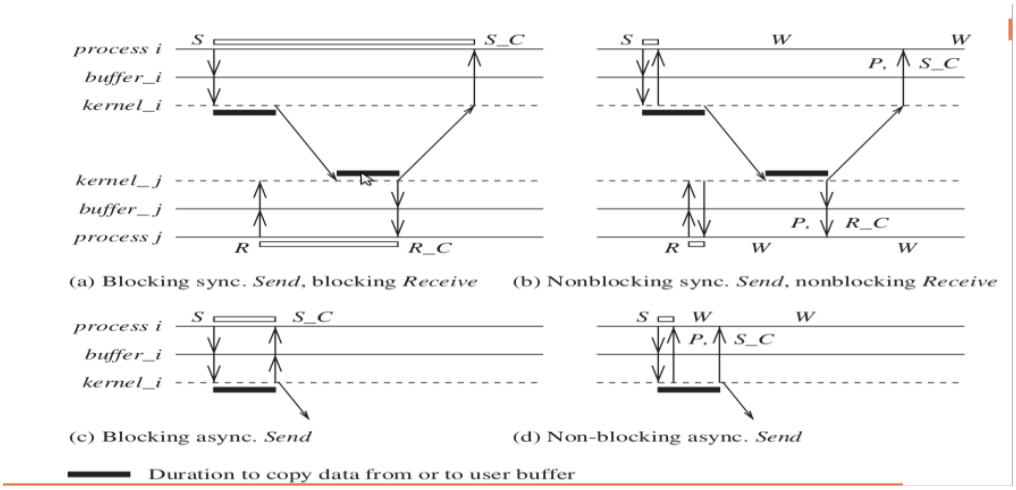


Figure 1.6: Timing diagram of Blocking/Non-blocking and Synchronous/Asynchronous Primitives

In the figure, three time lines are used for each processes:

1. for the process executions
2. for the user buffer from/to which data is sent/received
3. for the kernel/communication subsystem

• 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 acknowledgment 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 acknowledgment returns from the receiver, as per the semantics described in the blocking synchronous send. The user process can keep checking for the completion of the non-blocking synchronous Send by testing the returned handle, or it can invoke the blocking Wait operation on the returned handle

• 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. (For the unbuffered option, the user process that invokes the Send is blocked until the transfer of the data from the user's buffer to the network is initiated.) Control returns to the user process as soon as this transfer is initiated, and a parameter in the non-locking call also gets set with the handle of a location that the user process can check later 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. The checking for the completion may be necessary if the user wants to reuse the buffer from which the data was sent.

• 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. (If the data has already arrived when the call is made, it would be pending in some kernel buffer, and still needs to be copied to the user buffer.)

Observation:

Blocking synchronous Send and Blocking receive

A blocking synchronous Send is easier to use from a programmer's perspective because the handshake between the Send and the Receive makes the communication appear instantaneous, thereby simplifying the program logic.

A blocking receive Receive, may not get issued until much after the data arrives at P j , in which case the data arrived would have to be buffered in the system buffer at P j and not in the user buffer. At the same time, the sender would remain blocked.

"A blocking synchronous Send lowers the efficiency within process process Pi."

The ***non-blocking asynchronous Send*** is useful when a large data item is being sent because it allows the process to perform other instructions in parallel with the completion of the Send.

The ***non-blocking synchronous Send*** also avoids the potentially large delays for handshaking, particularly when the receiver has not yet issued the Receive call.

The ***non-blocking Receive*** is useful when a large data item is being received and/or when the sender has not yet issued the Send call, because it allows the process to perform other instructions in parallel with the completion of the Receive.

Note that if the data has already arrived, it is stored in the kernel buffer, and it may take a while to copy it to the user buffer specified in the Receive call.

For non-blocking calls, however, the burden on the programmer increases because he or she has to keep track of the completion of such operations in order to meaningfully reuse (write to or read from) the user buffers.

"Thus, conceptually, blocking primitives are easier to use."

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

Message-Passing Primitives:

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.

Socket primitives or socket-like transport layer primitives are invoked to call the procedure remotely. There exist many implementations of RPC.

For example:

Sun RPC, and distributed computing environment (DCE) RPC. "Messaging" and "streaming" are two other mechanisms for communication.

With the growth of object based software, libraries for remote method invocation (RMI) and remote object invocation (ROI) with their own set of primitives are being proposed and standardized by different agencies.

CORBA (common object request broker architecture) and DCOM (distributed component object model) are two other standardized architectures with their own set of primitives. Additionally, several projects in the research stage are designing their own flavour of communication primitives.

1.7 Synchronous versus Asynchronous Executions

A synchronous execution is an execution in which (i) processors are synchronized and the clock drift rate between any two processors is bounded.

It is easier to design and verify algorithms assuming synchronous executions because of the coordinated nature of the executions at all the processes. However, there is a hurdle to having a truly synchronous execution. It is practically difficult to build a completely synchronous system, and have the messages delivered within a bounded time. Therefore, this synchrony has to be simulated under the covers, and will inevitably involve delaying or blocking some processes for some time durations.

Thus, 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 sub- phases or sub-steps) that processes execute. Each sub-round is assumed to send at most one message per process; hence the message(s) sent will reach in a single message hop.

The timing diagram of an example synchronous execution, there are four nodes P 0 to P 3 . In each round, process P i sends a message to P #i+1# mod 4 and P #i−1# mod 4 and calculates some application-specific function on the received values.

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

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?

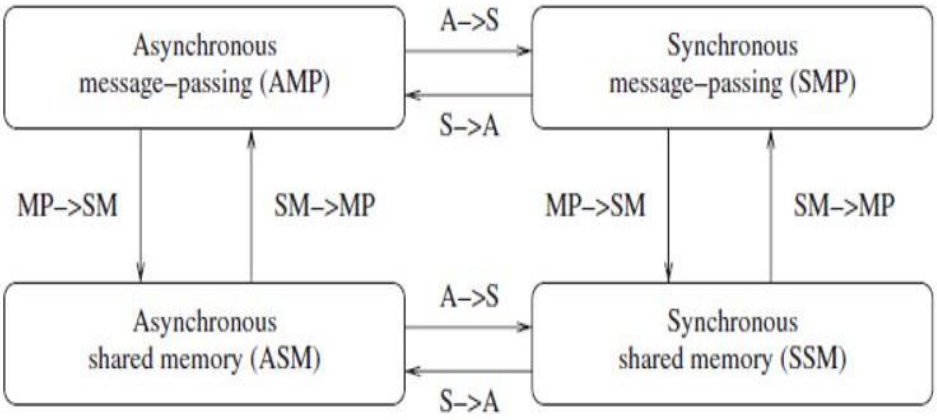


Figure 1.7: Emulation among the principal system classes in a failure-free system

1.8 Design issues and challenges

Distributed computing systems have been in widespread existence since the 1970s when the Internet and ARPANET came into being.

At the time, the primary issues in the design of the distributed systems included providing access to remote data in the face of failures, file system design, and directory structure design.

Categories of important Design Issues and Challenges are:

(i) The system community, which needs to having a greater component related to systems design and operating systems design.

OR

ii) The theoretical algorithms community within distributed computing, which needs to having a greater component related to algorithm design.

OR

(iii) The forces driving the emerging applications and technology, that is Emerging from recent technology advances and/or driven by new applications.
For example:

The current distributed computing follows the "Client-Server" architecture to a large degree, which needs the attention in the theoretical distributed algorithms community.

Two reason identified

(1) Simple adequate models are needed for the number of applications outside the scientific computing community of users in distributed systems are Business applications.

(2) The practice are largely controlled by Industry standards, not a technically best solution.

Distributed System Perspective Challenges

1. Communication

To design appropriate mechanisms for communication among the processes in the network.

For example:

RPC – Remote Procedure Call
ROI – Remote Object Invocation
Message-oriented communication Vs Stream-oriented communication.

2. Processes

Management of processes and threads at clients/servers
Code Migration
Design of Software
Mobile Agents.

3. Naming

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.

4. Synchronization

Mechanisms for synchronization or coordination among the processes are essential. Mutual exclusion is the classical exam- ple 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.

5. 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.

6. 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.

7. 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, check pointing and recovery, agreement and consensus, failure detection, and self-stabilization are some of the mechanisms to provide fault-tolerance.

8. Security

Distributed systems security involves various aspects of cryptography, secure channels, access control, key management generation and distribution, authorization, and secure group management.

9. 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:

Access transparency hides differences in data representation on different systems and provides uniform operations to access system resources.

Location transparency makes the locations of resources transparent to the users.

Relocation transparency having the ability to relocate the resources as they being accessed.

Migration transparency allows relocating resources without changing names.

Replication transparency does not let the user become aware of any replication.

Concurrency transparency deals with masking the concurrent use of shared resources for the user.

Failure transparency refers to the system being reliable and fault-tolerant.

10. 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

1. 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.

For example: Input / Output Automata Model and Temporal Logic of Actions (TLA)

2. Dynamic Distributed Graph Algorithms and Distributed Routing Algorithms

The graph algorithms form the building blocks for a large number of higher level communication, data dissemination, object location, and object search functions.

The algorithms need to deal with dynamically changing graph characteristics, such as to model varying link loads in a routing algorithm.

The efficiency of these algorithms impacts not only the user-perceived latency but also the traffic and hence the load or congestion in the network.

3. Time and Global State in a Distributed System

Two dimensions are concerned are: **Time and Space**.

The challenges pertain to providing accurate physical time, and to providing a variant of time, called logical time. Logical time is relative time, and eliminates the overheads of providing physical time for applications where physical time is not required. More importantly, 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.

Global state of the system (across space) also involves the time dimension for consistent observation. Due to the inherent distributed nature of the system, it is not possible for any one process to directly observe a meaningful global state across all the processes, without using extra state-gathering effort which needs to be done in a coordinated manner.

4. 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.

The synchronization mechanisms can also be viewed as ***resource management and concurrency management mechanisms***.

Examples of problems requiring synchronization:

1. Physical Clock Synchronization

Physical clocks usually diverge in their values due to hardware limitations. Keeping them synchronized is a fundamental challenge to maintain common time.

2. 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.

3. Mutual Exclusion

Access to the critical resource(s) has to be coordinated.

4. Deadlock Detection and Resolution

Deadlock detection should be coordinated to avoid duplicate work.

Deadlock resolution should be coordinated to avoid unnecessary aborts of processes.

5. Termination Detection

This requires cooperation among the processes to detect the specific global state of quiescence.

6. Garbage Collection

Garbage refers to objects that are no longer in use and that are not pointed to by any other process. Detecting garbage requires coordination among the processes.

5. 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. When multiple processes send messages concurrently, different recipients may receive the messages in different orders, possibly violating the semantics of the distributed program.

Hence, formal specifications of the semantics of ordered delivery need to be formulated, and then implemented.

6. 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. On-line algorithms for monitoring such predicates are hence important.

An important paradigm for monitoring distributed events is that of event streaming, wherein streams of relevant events reported from different processes are examined collectively to detect predicates.

Typically, the specification of such predicates uses physical or logical time relationships.

7. 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.

8. 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.

9. 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. Additionally, placement of the replicas in the systems is also a challenge because resources usually cannot be freely replicated.

10. 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. Further, prefetching of objects when access patterns and other characteristics of the objects are known, can also be performed.

An example of where prefetching can be used is the case of subscribing to Content Distribution Servers. Minimizing response time to minimize user- perceived latencies is an important challenge.

Object search and navigation on the web are important functions in the operation of the web, and are very resource-intensive.

Distributed Shared Memory Abstraction

A shared memory abstraction simplifies the task of the programmer to deal only with read and write operations, and no message communication primitives.

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.

a. Wait-free algorithms

Wait-free algorithms defined as the ability of a process to complete its execution irrespective of the actions of other processes, gained prominence in the design of algorithms to control access to shared resources in the shared memory abstraction.

It corresponds to n − 1-fault resilience in a n process system and is an important principle in fault-tolerant system design. While wait-free algorithms are highly desirable, they are also expensive, and designing low overhead wait-free algorithms is a challenge.

b. Mutual Exclusion

Operating systems covers the basic algorithms (such as the Bakery algorithm and using semaphores) for mutual exclusion in a multiprocessing (uniprocessor or multiprocessor) shared memory setting.

c. Register Constructions

Register constructions deals with the design of registers from scratch, with very weak assumptions on the accesses allowed to a register. This field forms a foundation for future architectures that allow concurrent access even to primitive units of memory (independent of technology) without any restrictions on the concurrency permitted.

d. Consistency Models

For multiple copies of a variable/object, varying degrees of consistency among the replicas can be allowed. These represent a trade-off of coherence versus cost of implementation. A strict definition of consistency (such as in a uniprocessor system) would be expensive to implement in terms of high latency, high message over- head, and low concurrency. Hence, relaxed but still meaningful models of consistency are desirable.

Load Balancing

The goal of load balancing is to gain higher throughput, and reduce the user perceived latency. Load balancing may be necessary because of a variety of factors such as high network traffic or high request rate causing the network connection to be a bottleneck, or high computational load.

Some form of load balancing are:

• Data migration

The ability to move data (which may be replicated) around in the system, based on the access pattern of the users.

• ***Computation migration***

The ability to relocate processes in order to perform a redistribution of the workload.

• ***Distributed scheduling***

This achieves a better turnaround time for the users by using idle processing power in the system more efficiently.

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.

Message propagation delays which are network-dependent are hard to control or predict, which makes meeting real-time guarantees that are inherently dependent on communication among the processes harder.

Performance

High throughput is not the primary goal of using a distributed system, achieving good performance is important.

In large distributed systems, network latency (propagation and transmission times) and access to shared resources can lead to large delays which must be minimized. The user perceived turn-around time is very important.

Some example issues arise in determining the performance are:

• **Metrics**

Appropriate metrics must be defined or identified for measuring the performance of theoretical distributed algorithms, as well as for implementations of such algorithms. The former would involve various complexity measures on the metrics, whereas the latter would involve various system and statistical metrics.

• **Measurement methods/tools**

As a real distributed system is a complex entity and has to deal with all the difficulties that arise in measuring performance over a WAN/the Internet, appropriate methodologies and tools must be developed for measuring the performance metrics.

Applications of Distributed Computing and Newer Challenges

1. Mobile Systems

Mobile systems typically use wireless communication which is based on electromagnetic waves and utilizes a shared broadcast medium.

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.

From a computer science perspective, there is a rich set of problems such as routing, location management, channel allocation, localization and position estimation, and the overall management of mobility.

There are two popular architectures: (a) Cellular Approach (b) Ad-hoc Network Approach

(a) Cellular Approach (base-station approach)

A cell 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.

(b) Ad-hoc Network Approach

In this approach, there is no base station (a centralized node acted as 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.

It is a complex model and it poses many graph-theoretical challenges from computer science and various engineering challenges.

2. 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 10^6 or higher) low-cost sensors.

The streaming data reported from a sensor network differs from the streaming data reported by "computer processes" in that the events reported by a sensor network are in the environment, external to the computer network and processes.

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.

3. 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,

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. Such systems are typically characterized as having many small processors operating collectively in a dynamic ambient network. The processors may be connected to more powerful networks and processing resources in the background for processing and collating data.

4. 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. No central directories (such as those used in domain name servers) for name resolution and object lookup are allowed.

Some of the key challenges include: object storage mechanisms, efficient object lookup, and retrieval in a scalable manner; dynamic reconfiguration with nodes as well as objects joining and leaving the network randomly; replication strategies to expedite object search; trade-offs between object size latency and table sizes; anonymity, privacy, and security.

5. Publish-Subscribe, Content Distribution, and Multimedia

In a dynamic environment where the information constantly fluctuates.
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.

Content distribution refers to a class of mechanisms, primarily in the web and P2P computing context, whereby specific information which can be broadly characterized by a set of parameters is to be distributed to interested processes.

Clearly, there is overlap between content distribution mechanisms and publish–subscribe mechanisms. When the content involves multimedia data, special requirement such as the following arise: multimedia data is usually very large and information-intensive, requires compression, and often requires special synchronization during storage and playback.

6. 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. Agents collect and process information, and can exchange such information with other agents.

Challenges in distributed agent systems include coordination mechanisms among the agents, controlling the mobility of the agents, and their software design and interfaces.

Research in agents is inter-disciplinary:

Spanning artificial intelligence, mobile computing, economic market models, software engineering, and distributed computing.

7. 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. The mining can be done by applying database and artificial intelligence techniques to a data repository.

The data is necessarily distributed and cannot be collected in a single repository, as in banking applications.

Challenges in data mining include, efficient distributed data mining algorithms are required.

8. Grid Computing

Idle CPU cycles of machines connected to the network will be avail- able to others.

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.

9. Security in Distributed Systems

The traditional challenges of security in a distributed setting include:

Confidentiality: Ensuring that only authorized processes can access certain information. A***uthentication:*** Ensuring the source of received information and the identity of the sending process ***Availability:*** Maintaining allowed access to services despite malicious actions.

The goal is to meet these challenges with efficient and scalable solutions.

For the newer distributed architectures, such as wireless, peer-to-peer, grid, and pervasive computing discussed in this sub- section), these challenges become more interesting due to factors

such as a resource-constrained environment, a broadcast medium, the lack of structure, and the lack of trust in the network.

1.9 Summary

Resource sharing is the main motivating factor for constructing distributed systems. Resources such as printers, files, web pages or database records are managed by servers of the appropriate type.

For example, web servers manage web pages and other web resources. Resources are accessed by clients – for example, the clients of web servers are generally called browsers.

The construction of distributed systems produces many challenges:

Heterogeneity: They must be constructed from a variety of different networks, operating systems, computer hardware and programming languages. The Internet communication protocols mask the difference in networks, and middleware can deal with the other differences.

Openness: Distributed systems should be extensible – the first step is to publish the interfaces of the components, but the integration of components written by different programmers is a real challenge.

Security: Encryption can be used to provide adequate protection of shared resources and to keep sensitive information secret when it is transmitted in messages over a network. Denial of service attacks are still a problem.

Scalability: A distributed system is scalable if the cost of adding a user is a constant amount in terms of the resources that must be added. The algorithms used to access shared data should avoid performance bottlenecks and data should be structured hierarchically to get the best access times. Frequently accessed data can be replicated.

Failure handling: Any process, computer or network may fail independently of the others. Therefore each component needs to be aware of the possible ways in which the components it depends on may fail and be designed to deal with each of those failures appropriately.

Concurrency: The presence of multiple users in a distributed system is a source of concurrent requests to its resources. Each resource must be designed to be safe in a concurrent environment.

Transparency: The aim is to make certain aspects of distribution invisible to the application programmer so that they need only be concerned with the design of their particular application. For example, they need not be concerned with its location or the details of how its operations are accessed by other components, or whether it will be replicated or migrated. Even failures of networks and processes can be presented to application programmers in the form of exceptions – but they must be handled.

Quality of service: It is not sufficient to provide access to services in distributed systems. In particular, it is also important to provide guarantees regarding the qualities associated with such service access. Examples of such qualities include parameters related to performance, security and reliability.