CHAPTER

1 Introduction

1.1 Definition

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. Distributed systems have been in existence since the start of the universe. From a school of fish to a flock of birds and entire ecosystems of microorganisms, there is communication among mobile intelligent agents in nature. With the widespread proliferation of the Internet and the emerging global village, the notion of distributed computing systems as a useful and widely deployed tool is becoming a reality. For computing systems, a distributed system has been characterized in one of several ways:

- You know you are using one when the crash of a computer you have never heard of prevents you from doing work [23].
- A collection of computers that do not share common memory or a common physical clock, that communicate by a messages passing over a communication network, and where each computer has its own memory and runs its own operating system. Typically the computers are semi-autonomous and are loosely coupled while they cooperate to address a problem collectively [29].
- A collection of independent computers that appears to the users of the system as a single coherent computer [33].
- A term that describes a wide range of computers, from weakly coupled systems such as wide-area networks, to strongly coupled systems such as local area networks, to very strongly coupled systems such as multiprocessor systems [19].

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.

  It may be noted that a distributed system may still provide the abstraction of a common address space via the distributed shared memory abstraction. Several aspects of shared memory multiprocessor systems have also been studied in the distributed computing literature.
- **Geographical separation** The geographically wider apart that the processors are, the more representative is the system of a distributed system. However, it is not necessary for the processors to be on a wide-area network (WAN). Recently, the network/cluster of workstations (NOW/COW) configuration connecting processors on a LAN is also being increasingly regarded as a small distributed system. This NOW configuration is becoming popular because of the low-cost high-speed off-the-shelf processors now available. The Google search engine is based on the NOW architecture.
- **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*.

The distributed system uses a layered architecture to break down the complexity of system design. The middleware is the distributed software that

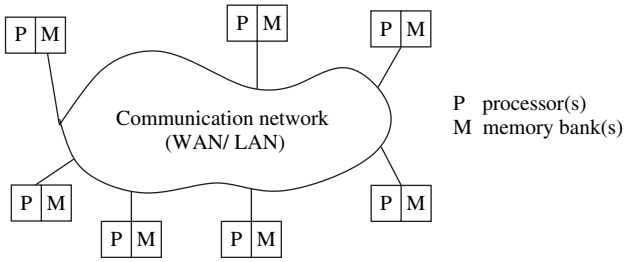


Figure 1.1 A distributed system connects processors by a communication network.

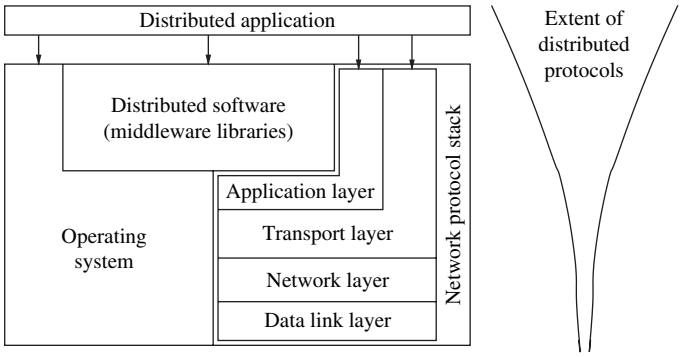


Figure 1.2 Interaction of the software components at each processor.

drives the distributed system, while providing transparency of heterogeneity at the platform level [24]. Figure 1.2 schematically shows the interaction of this software with these system components at each processor. Here we 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 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 multicasting – of the middleware layer. There are several standards such as Object Management Group's (OMG) common object request broker architecture (CORBA) [36], and the remote procedure call (RPC) mechanism [1, 11]. 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) [7] technologies. The message-passing interface (MPI) [20, 30] 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. 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, i.e., the resource should be accessible at all times;
   - integrity, i.e., the value/state of the resource should be correct, in the face of concurrent access from multiple processors, as per the semantics expected by the application;
   - fault-tolerance, i.e., the ability to recover from system failures, where such failures may be defined to occur in one of many failure models, which we will study in Chapters 5 and 14.
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 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 Relation to parallel multiprocessor/multicomputer systems

The characteristics of a distributed system were identified above. A typical distributed system would look as shown in Figure 1.1. However, how does one classify a system that meets some but not all of the characteristics? Is the system still a distributed system, or does it become a parallel multiprocessor system? To better answer these questions, we first examine the architecture of parallel systems, and then examine some well-known taxonomies for multiprocessor/multicomputer systems.

1.4.1 Characteristics of parallel systems

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

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. The architecture is shown in Figure 1.3(a). 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. Interprocess 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

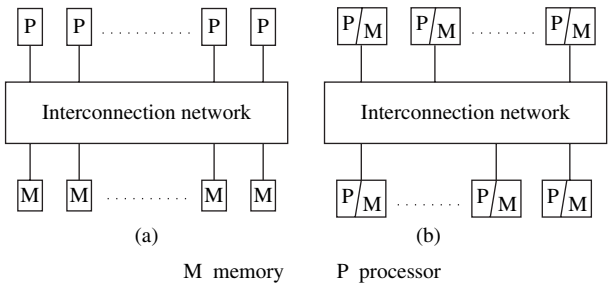
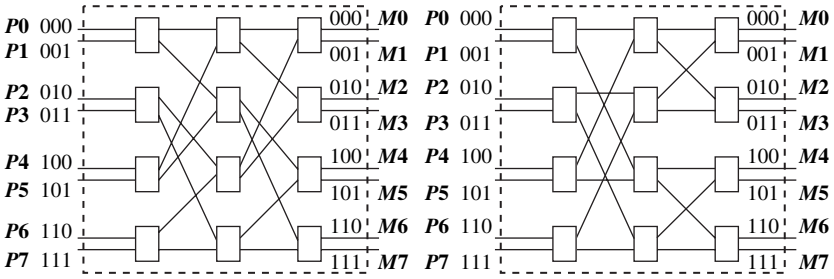


Figure 1.3 Two standard architectures for parallel systems. (a) Uniform memory access (UMA) multiprocessor system. (b) Non-uniform memory access (NUMA) multiprocessor. In both architectures, the processors may locally cache data from memory.



(a) 3-stage Omega network ($n = 8$, $M = 4$) (b) 3-stage Butterfly network ($n = 8$, $M = 4$)

Figure 1.4 Interconnection networks for shared memory multiprocessor systems. (a) Omega network [4] for $n = 8$ processors $P0$–$P7$ and memory banks $M0$–$M7$. (b) Butterfly network [10] for $n = 8$ processors $P0$–$P7$ and memory banks $M0$–$M7$.

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. 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.

Figure 1.4 shows two popular interconnection networks – the Omega network [4] and the Butterfly network [10], each of which is a multi-stage network formed of 2×2 switching elements. Each 2×2 switch allows data on either of the two input wires to be switched to the upper or the lower output wire. In a single step, however, only one data unit can be sent on an 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. Furthermore, 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. Besides the Omega and Butterfly (banyan) networks, other examples of multistage interconnection networks are the Clos [9] and the shuffle-exchange networks [37]. Each of these has very interesting mathematical properties that allow rich connectivity between the processor bank and memory bank.

Omega interconnection function The Omega network which connects n processors to n memory units has $n/2 log_2 n$ 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:

$$j = \begin{cases} 2i, & \text{for } 0 \leq i \leq n/2 - 1, \\ 2i + 1 - n, & \text{for } n/2 \leq i \leq n - 1. \end{cases} \tag{1.1}$$

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 $0 \leq i \leq 3$, the output i is connected to input $2i$ of the next stage. For $4 \leq i \leq 7$, 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 $s \in [0, log_2 n - 1]$. In a stage s switch, if the $s+1$th 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.

Butterfly interconnection function Unlike the Omega 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 expression is as follows. Let there be $M = n/2$ switches per stage, and let a switch be denoted by the tuple $\langle x, s \rangle$, where $x \in [0, M - 1]$ and stage $s \in [0, log_2 n - 1]$.

The two outgoing edges from any switch $\langle x, s \rangle$ are as follows. There is an edge from switch $\langle x, s \rangle$ to switch $\langle y, s + 1 \rangle$ 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.

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. (Note that unlike the Omega network formulation using input and output lines given above, this formulation uses switch numbers. Exercise 1.5 asks you to prove a formulation of the Omega interconnection pattern using switch numbers instead of input and output port numbers.)

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.

Observe that 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 architecture is shown in Figure 1.3(b).

   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(a) shows a wrap-around 4×4 mesh. For a $k \times k$ 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. Figure 1.5(b) shows a four-dimensional hypercube. A k-dimensional hypercube has 2^k processor-and-memory units [13,21]. 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 labelled 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.

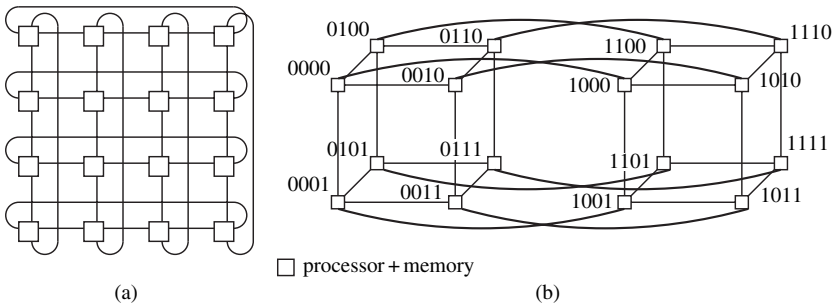


Figure 1.5 Some popular topologies for multicomputer shared-memory machines. (a) Wrap-around 2D-mesh, also known as torus. (b) Hypercube of dimension 4.

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. 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. This class of parallel systems has a very niche market.

The distinction between UMA multiprocessors on the one hand, and NUMA and message-passing multicomputers on the other, is important because the algorithm design and data and task partitioning among the processors must account for the variable and unpredictable latencies in accessing memory/communication [22]. As compared to UMA systems and array processors, NUMA and message-passing multicomputer systems are less suitable when the degree of granularity of accessing shared data and communication is very fine.

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 subtasks 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. Searching through large state spaces can be performed with significant speedup on parallel machines. While such parallel machines were an object of much theoretical and systems research in the 1980s and 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 [14] 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 [22].
- **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.