Chapter-1

Introduction to Distributed Systems

1.1 Definition and Characteristics

The pace at which computer systems change was, is, and continues to be overwhelming. From 1945, when the modern computer era began, until about 1985, computers were large and expensive. Moreover, for lack of a way to connect them, these computers operated independently from one another.

Starting in the mid-1980s, however, two advances in technology began to change that situation. The first was the development of powerful microprocessors. Initially, these were 8-bit machines, but soon 16-, 32-, and 64-bit CPUs became common. The current generation of machines have the computing power of the mainframes deployed 30 or 40 years ago, but for 1/1000th of the price or less.

The second development was the invention of high-speed computer networks. Local-area networks or LANs allow thousands of machines within a building to be connected in such a way that small amounts of information can be transferred in a few microseconds or so. Larger amounts of data can be moved between machines at rates of billions of bits per second (bps).

Parallel to the development of increasingly powerful and networked machines, we have also been able to witness miniaturization of computer systems with perhaps the smartphone as the most impressive outcome.

Packed with sensors, lots of memory, and a powerful CPU, these devices are nothing less than full-fledged computers. Of course, they also have networking capabilities.

Along the same lines, so-called plug computers are finding their way to the market. These small computers, often the size of a power adapter, can be plugged directly into an outlet and offer near-desktop performance.

The result of these technologies is that it is now not only feasible, but easy, to put together a computing system composed of a large numbers of networked computers, be they large or small. These computers are generally geographically dispersed, for which reason they are usually said to form a distributed system. The size of a distributed system may vary from a handful of devices, to millions of computers.

The interconnection network may be wired, wireless, or a combination of both. Moreover, distributed systems are often highly dynamic, in the sense that computers can join and leave, with the topology and performance of the underlying network almost continuously changing.
Various definitions of distributed systems have been given in the literature, none of them satisfactory, and none of them in agreement with any of the others. For our purposes it is sufficient to give a loose characterization:

A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system.

This definition refers to two characteristic features of distributed systems.

Characteristic 1: Collection of autonomous computing elements
A computing element, which we will generally refer to as a node, can be either a hardware device or a software process.

Modern distributed systems can, and often will, consist of all kinds of nodes, ranging from very big high-performance computers to small plug computers or even smaller devices.

A fundamental principle is that nodes can act independently from each other, although it should be obvious that if they ignore each other, then there is no use in putting them into the same distributed system. In practice, nodes are programmed to achieve common goals, which are realized by exchanging messages with each other.

Characteristic 2: Single coherent system
This means that one way or another the autonomous nodes need to collaborate. How to establish this collaboration lies at the heart of developing distributed systems. Note that we are not making any assumptions concerning the type of nodes. Likewise, we make no assumptions concerning the way that nodes are interconnected.

Roughly speaking, a distributed system is coherent if it behaves according to the expectations of its users. More specifically, in a single coherent system the collection of nodes as a whole operates the same, no matter where, when, and how interaction between a user and the system takes place.

Fundamental concepts in DS
Execution clock
An important observation is that, as a consequence of dealing with independent nodes, each one will have its own notion of time. In other words, we cannot always assume that there is something like a global clock. This lack of a common reference of time leads to fundamental questions regarding the synchronization and coordination within a distributed system.

Membership issues
The fact that we are dealing with a collection of nodes implies that we may also need to manage the membership and organization of that collection. In other words, we may need to register which nodes may or may not belong to the system, and also provide each member with a list of nodes it can directly communicate with.

Managing group membership can be exceedingly difficult, if only for reasons of admission control. To explain, we make a distinction between open and closed groups. In an open group, any node is allowed to join the distributed system, effectively meaning that it can send messages to any other

node in the system. In contrast, with a closed group, only the members of that group can communicate with each other and a separate mechanism is needed to let a node join or leave the group.

Authentication issues
It is not difficult to see that admission control can be difficult. First, a mechanism is needed to authenticate a node, and if not properly designed, managing authentication can easily create a scalability bottleneck. Second, each node must, in principle, check if it is indeed communicating with another group member and not, for example, with an intruder aiming to create havoc. Finally, considering that a member can easily communicate with nonmembers, if confidentiality is an issue in the communication within the distributed system, we may be facing trust issues

Node organization and communication issues
Concerning the organization of the collection, practice shows that a distributed system is often organized as an overlay network. In this case, a node is typically a software process equipped with a list of other processes it can directly send messages to. It may also be the case that a neighbor needs to be first looked up. Message passing is then done through TCP/IP or UDP channels.

There are roughly two types of overlay networks:
Structured overlay: In this case, each node has a well-defined set of neighbors with whom it can communicate. For example, the nodes are organized in a tree or logical ring.

Unstructured overlay: In these overlays, each node has a number of references to randomly selected other nodes.

In any case, an overlay network should, in principle, always be connected, meaning that between any two nodes there is always a communication path allowing those nodes to route messages from one to the other. A well-known class of overlays is formed by peer-to-peer (P2P) networks.

It is important to realize that the organization of nodes requires special effort and that it is sometimes one of the more intricate parts of distributed-systems management.

Middleware and distributed systems
To assist the development of distributed applications, distributed systems are often organized to have a separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system. This organization is shown in Figure 1.1, leading to what is known as middleware

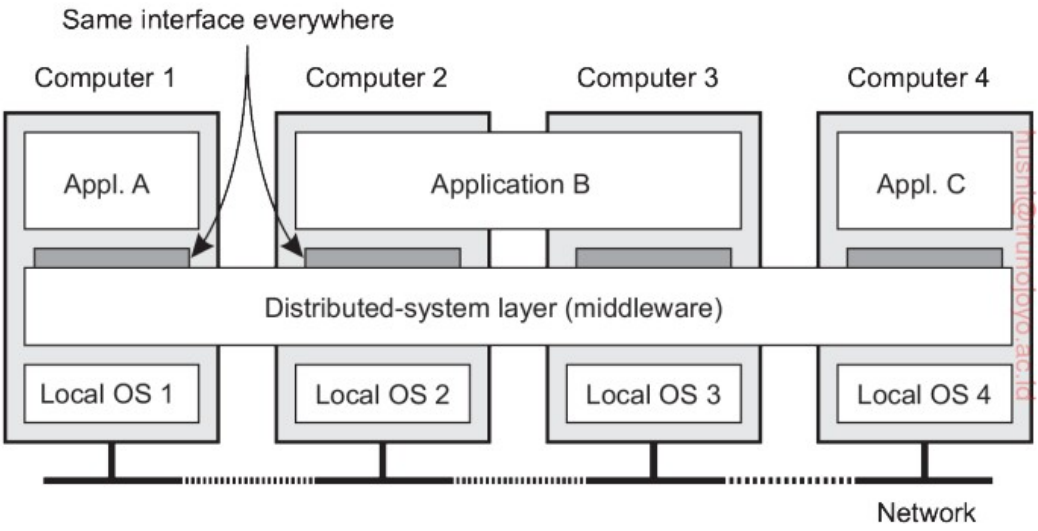

Figure 1.1: A distributed system organized in a middleware layer, which extends over multiple machines, offering each application the same interface.

Figure 1.1 shows four networked computers and three applications, of which application B is distributed across computers 2 and 3. Each application is offered the same interface. The distributed system provides the means for components of a single distributed application to communicate with each other, but also to let different applications communicate. At the same time, it hides, as best and reasonably as possible, the differences in hardware and operating systems from each application.

In a sense, middleware is the same to a distributed system as what an operating system is to a computer: a manager of resources offering its applications to efficiently share and deploy those resources across a network. Next to resource management, it offers services that can also be found in most operating systems, including:

I. Facilities for interapplication communication.
II. Security services.
III. Accounting services.
IV. Masking of and recovery from failures

The main difference with their operating-system equivalents, is that middleware services are offered in a networked environment.

1.3 Organization and goals of distributed systems

Just because it is possible to build distributed systems does not necessarily mean that it is a good idea. In this section we discuss four important goals that should be met to make building a distributed system worth the effort. A distributed system should make resources easily accessible; it should hide the fact that resources are distributed across a network; it should be open; and itshould be scalable.

I. Supporting resource sharing

An important goal of a distributed system is to make it easy for users (and applications) to access and share remote resources. Resources can be virtually anything, but typical examples include peripherals, storage facilities, data, files, services, and networks, to name just a few. There are many reasons for wanting to share resources. One obvious reason is that of economics. For example, it is cheaper to have a single high-end reliable storage facility be shared than having to buy and maintain storage for each user separately.

Connecting users and resources also makes it easier to collaborate and exchange information, as is illustrated by the success of the Internet with its simple protocols for exchanging files, mail, documents, audio, and video.

II. Making distribution transparent

An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers possibly separated by large distances. In other words, it tries to make the distribution of processes and resources transparent, that is, invisible, to end users and applications.

Types of distribution transparency

The concept of transparency can be applied to several aspects of a distributed system, of which the most important ones are listed in Figure 1.2. We use the term object to mean either a process or a resource.

Transparency	**Description**
Access	Hide differences in data representation and how an object is accessed
Location	Hide where an object is located
Relocation	Hide that an object may be moved to another location while in use
Migration	Hide that an object may move to another location
Replication	Hide that an object is replicated
Concurrency	Hide that an object may be shared by several independent users
Failure	Hide the failure and recovery of an object

Figure 1.2: Different forms of transparency in a distributed system (see ISO [1995]). An object can be a resource or a process.

III. Being open

Another important goal of distributed systems is openness. An open distributed system is essentially a system that offers components that can easily be used by, or integrated into other systems. At the same time, an open distributed system itself will often consist of components that originate from elsewhere.

Interoperability, composability, and extensibility

Interoperability characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard.

Portability characterizes to what extent an application developed for a distributed system A can be executed, without modification, on a different distributed system B that implements the same interfaces as A.

Another important goal for an open distributed system is that it should be easy to configure the system out of different components (possibly from different developers). Also, it should be easy to add new components or replace existing ones without affecting those components that stay in place. In other words, an open distributed system should also be extensible

IV. Being scalable

For many of us, worldwide connectivity through the Internet is as common as being able to send a postcard to anyone anywhere around the world. Moreover, where until recently we were used to having relatively powerful desktop computers for office applications and storage, we are now witnessing that such applications and services are being placed in what has been coined "the cloud," in turn leading to an increase of much smaller networked devices such as tablet computers. With this in mind, scalability has become one of the most important design goals for developers of distributed systems.

Scalability dimensions

Scalability of a system can be measured along at least three different dimensions.

Size scalability: A system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system without any noticeable loss of performance.

Geographical scalability: A geographically scalable system is one in which the users and resources may lie far apart, but the fact that communication delays may be significant is hardly noticed.

Administrative scalability: An administratively scalable system is one that can still be easily managed even if it spans many independent administrative organizations.

1.4 The client-server model

In the basic client-server model, processes in a distributed system are divided into two (possibly overlapping) groups. A server is a process implementing a specific service, for example, a file system service or a database service. A client is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply. This client-server interaction, also known as request-reply behavior is shown in Figure 1.3 in the form of a message sequence chart.

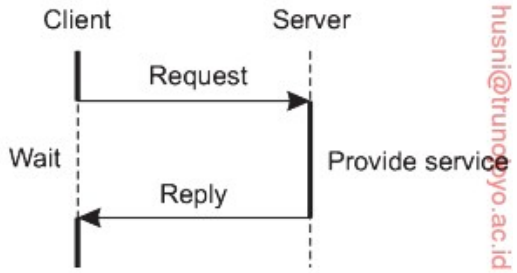

Figure 1.3: General interaction between a client and a server.

Communication between a client and a server can be implemented by means of a simple connectionless protocol when the underlying network is fairly reliable as in many local-area networks. In these cases, when a client requests a service, it simply packages a message for the server, identifying the service it wants, along with the necessary input data. The message is then sent to the server. The latter, in turn, will always wait for an incoming request subsequently process it, and package the results in a reply message that is then sent to the client.

Using a connectionless protocol has the obvious advantage of being efficient. As long as messages do not get lost or corrupted, the request/reply protocol just sketched works fine. Unfortunately, making the protocol resistant to occasional transmission failures is not trivial. The only thing we can do is possibly let the client resend the request when no reply message comes in. The problem, however, is that the client cannot detect whether the original request message was lost, or that transmission of the reply failed. If the reply was lost, then resending a request may result in performing the operation twice. If the operation was something like "transfer $10,000 from my bank account," then clearly, it would have been better that we simply reported an error instead. On the other hand, if the operation was "tell me how much money I have left," it would be perfectly acceptable to resend the request. When an operation can be repeated multiple times without harm, it is said to be idempotent. Since some requests are idempotent and others are not it should be clear that there is no single solution for dealing with lost messages

As an alternative, many client-server systems use a reliable connection-oriented protocol. Although this solution is not entirely appropriate in a local-area network due to relatively low performance, it works perfectly fine in wide-area systems in which communication is inherently unreliable. For example, virtually all Internet application protocols are based on reliable TCP/IP connections. In this case, whenever a client requests a service, it first sets up a connection to the

server before sending the request. The server generally uses that same connection to send the reply message, after which the connection is torn down. The trouble may be that setting up and tearing down a connection is relatively costly, especially when the request and reply messages are small.

The client-server model has been subject to many debates and controversies over the years. One of the main issues was how to draw a clear distinction between a client and a server. Not surprisingly, there is often no clear distinction. For example, a server for a distributed database may continuously act as a client because it is forwarding requests to different file servers responsible for implementing the database tables. In such a case, the database server itself only processes the queries.

Multitiered Architectures

The distinction into three logical levels as discussed so far, suggests a number of possibilities for physically distributing a client-server application across several machines. The simplest organization is to have only two types of machines:

1. A client machine containing only the programs implementing (part of) the user-interface level
2. A server machine containing the rest, that is, the programs implementing the processing and data level

In this organization everything is handled by the server while the client is essentially no more than a dumb terminal, possibly with only a convenient graphical interface. There are, however, many other possibilities, many distributed applications are divided into the three layers (1) user interface layer, (2) processing layer, and (3) data layer.

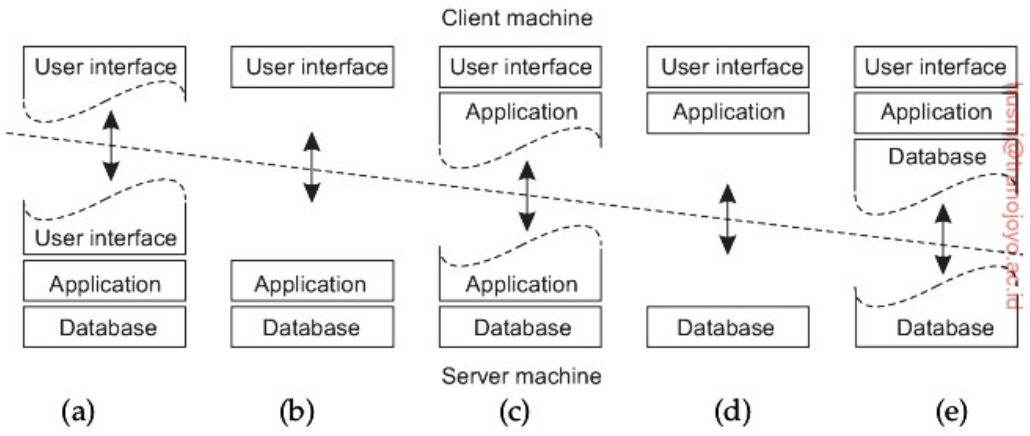

Figure 1.4: Client-server organizations in a two-tiered architecture.

One approach for organizing clients and servers is then to distribute these layers across different machines, as shown in Figure 1.4. As a first step, we make a distinction between only two kinds of machines: client machines and server machines, leading to what is also referred to as a (physically) two-tiered architecture.

One possible organization is to have only the terminal-dependent part of the user interface on the client machine, as shown in Figure 1.4(a), and give the applications remote control over the presentation of their data. An alternative is to place the entire user-interface software on the client side, as shown in Figure 1.4(b). In such cases, we essentially divide the application into a graphical front end, which communicates with the rest of the application (residing at the server) through an application-specific protocol. In this model, the front end (the client software) does no processing other than necessary for presenting the application's interface.

Continuing along this line of reasoning, we may also move part of the application to the front end, as shown in Figure 1.4(c).

In many client-server environments, the organizations shown in Figure 1.4(d) and Figure 1.4(e) are particularly popular. These organizations are used where the client machine is a PC or workstation, connected through a network to a distributed file system or database. Essentially, most of the application is running on the client machine, but all operations on files or database entries go to the server.