## 1.1 DEFINITION OF A DISTRIBUTED SYSTEM

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 independent computers that appears to its users as a single coherent system.*

This definition has two aspects. The first one deals with hardware: the machines are autonomous. The second one deals with software: the users think they are dealing with a single system. Both are essential. We will come back to these points later in this chapter after going over some background material on both the hardware and the software.

Instead of going further with definitions, it is perhaps more useful to concentrate on important characteristics of distributed systems. One important characteristic is that differences between the various computers and the ways in which they communicate are hidden from users. The same holds for the internal organization of the distributed system. Another important characteristic is that users and applications can interact with a distributed system in a consistent and uniform way, regardless of where and when interaction takes place.

Distributed systems should also be relatively easy to expand or scale. This characteristic is a direct consequence of having independent computers, but at the same time, hiding how these computers actually take part in the system as a whole. A distributed system will normally be continuously available, although perhaps certain parts may be temporarily out of order. Users and applications should not notice that parts are being replaced or fixed, or that new parts are added to serve more users or applications.

To support heterogeneous computers and networks while offering a single-system view, distributed systems are often organized by means of a layer of software that is logically placed between a higher-level layer consisting of users and

applications, and a layer underneath consisting of operating systems, as shown in Fig. 1-1 Accordingly, such a distributed system is sometimes called **middleware**.

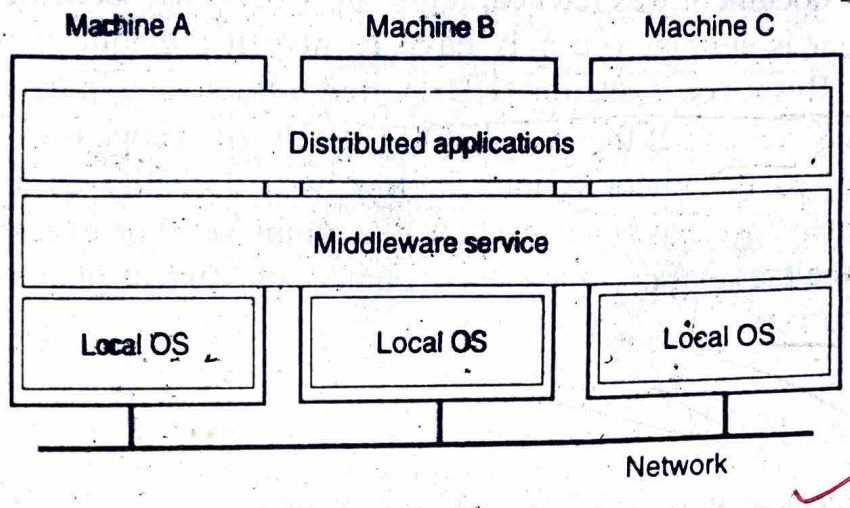


**Figure 1-1.** A distributed system organized as middleware. Note that the middleware layer extends over multiple machines.

Let us now take a look at several examples of distributed systems. As a first example, consider a network of workstations in a university or company department. In addition to each user's personal workstation, there might be a pool of processors in the machine room that are not assigned to specific users but are allocated dynamically as needed. Such a system might have a single file system, with all files accessible from all machines in the same way and using the same path name. Furthermore, when a user types a command, the system could look for the best place to execute that command, possibly on the user's own workstation, possibly on an idle workstation belonging to someone else, and possibly on one of the unassigned processors in the machine room. If the system as a whole looks and acts like a classical single-processor timesharing system (i.e., multi-user), it qualifies as a distributed system.

As a second example, consider a workflow information system that supports the automatic processing of orders. Typically, such a system is used by people from several departments, possibly at different locations. For example, people from the sales department may be spread across a large region or an entire country. Orders are placed by means of laptop computers that are connected to the system through the telephone network, possibly using cellular phones. Incoming orders are automatically forwarded to the planning department, resulting in new internal shipping orders sent to the stock department, as well as billing orders to be handled by the accounting department. The system will automatically forward orders to an appropriate and available person. Users are totally unaware of how orders physically flow through the system; to them it appears as if they are all operating on a centralized database.

As a final example, consider the World Wide Web. The Web offers a simple, consistent, and uniform model of distributed documents. To see a do...

user need merely activate a reference, and the document appears on the screen. In theory (but definitely not in current practice) there is no need to know from which server the document was fetched, let alone where that server is located. Publishing a document is simple: you only have to give it a unique name in the form of a **Uniform Resource Locator (URL)** that refers to a local file containing the document's content. If the World Wide Web would appear to its users as a gigantic centralized document system, it, too, would qualify as a distributed system. Unfortunately, we have not reached that point yet. For example, users are made aware of the fact that documents are located at different places and are handled by different servers.

## 1.2 GOALS

Just because it is possible to build distributed systems does not necessarily mean that it is a good idea. After all, with current technology it is also possible to put four floppy disk drives on a personal computer. It is just that doing so would be pointless. 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 easily connect users to resources; it should hide the fact that resources are distributed across a network; it should be open; and it should be scalable.

### 1.2.1 Connecting Users and Resources

The main goal of a distributed system is to make it easy for users to access remote resources, and to share them with other users in a controlled way. Resources can be virtually anything, but typical examples include printers, computers, storage facilities, data, files, Web pages, 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 let a printer be shared by several users than having to buy and maintain a separate printer for each user. Likewise, it makes sense to share costly resources such as supercomputers and high-performance storage systems.

Connecting users and resources also makes it easier to collaborate and exchange information, as is best illustrated by the success of the Internet with its simple protocols for exchanging files, mail, documents, audio, and video. The connectivity of the Internet is now leading to numerous virtual organizations in which geographically widely dispersed groups of people work together by means of **groupware**, that is, software for collaborative editing, teleconferencing, and so on. Likewise, the Internet connectivity has enabled electronic commerce allowing us to buy and sell all kinds of goods without actually having to go to a store.

However, as connectivity and sharing increase, security is becoming more and more important. In current practice, systems provide little protection against eavesdropping or intrusion on communication. Passwords and other sensitive

information are often sent as cleartext (i.e., unencrypted) through the network, or stored at servers that we can only hope are trustworthy. In this sense, there is much room for improvement. For example, it is currently possible to order goods by merely supplying a credit card number. Rarely is proof required that the customer owns the card. In the future, placing orders this way may be possible only if you can actually prove you physically possess the card by using a card reader.

Another security problem is that of tracking communication to build up a preference profile of a specific user (Wang et al., 1998). Such tracking explicitly violates privacy, especially if it is done without notifying the user. A related problem is that increased connectivity can also lead to unwanted communication, such as electronic junk mail often called spam. In such cases, what we may need is to protect ourselves using special information filters that select incoming messages based on their content.

### 1.2.2 Transparency

An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers. A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be **transparent**. Let us first take a look at what kinds of transparency exist in distributed systems, and then address the question whether transparency is always required.

#### Transparency in a Distributed System

The concept of transparency can be applied to several aspects of a distributed system, as shown in Fig. 1-2.

| Transparency | Description |
|---|---|
| Access | Hide differences in data representation and how a resource is accessed |
| Location | Hide where a resource is located |
| Migration | Hide that a resource may move to another location |
| Relocation | Hide that a resource may be moved to another location while in use |
| Replication | Hide that a resource is replicated |
| Concurrency | Hide that a resource may be shared by several competitive users |
| Failure | Hide the failure and recovery of a resource |
| Persistence | Hide whether a (software) resource is in memory or on disk |

**Figure 1-2.** Different forms of transparency in a distributed system (ISO, 1995).

**Access transparency** deals with hiding differences in data representation and the way that resources can be accessed by users. For example, to send an integer

from an Intel-based workstation to a Sun SPARC machine requires that we take into account that Intel orders its bytes in little endian format (i.e., the high-order byte is transmitted first), and that the SPARC processor uses **big endian** format (i.e., the low-order byte is transmitted first). Other differences in data representation may exist as well. For example, a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions. Differences in naming conventions, as well as how files can be manipulated, should all be hidden from users and applications.

An important group of transparency types has to do with the location of a resource. **Location transparency** refers to the fact that users cannot tell where a resource is physically located in the system. Naming plays an important role in achieving location transparency. In particular, location transparency can be achieved by assigning only logical names to resources, that is, names in which the location of a resource is not secretly encoded. An example of a such a name is the URL *http://www.prenhall.com/index.html*, which gives no clue about the location of Prentice Hall's main Web server. The URL also gives no clue as to whether *index.html* has always been at its current location or was recently moved there. Distributed systems in which resources can be moved without affecting how that resource can be accessed are said to provide **migration transparency**. Even stronger is the situation in which resources can be relocated *while* they are being accessed without the user or application noticing anything. In such cases, the system is said to support **relocation transparency**. An example of relocation transparency is when mobile users can continue to use their wireless laptop while moving from place to place without ever being (temporarily) disconnected.

As we shall see, replication plays an important role in distributed systems. For example, resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed. **Replication transparency** deals with hiding the fact that several copies of a resource exist. To hide replication from users, it is necessary that all replicas have the same name. Consequently, a system that supports replication transparency should generally support location transparency as well, because it would otherwise be impossible to refer to replicas at different locations.

We already mentioned that an important goal of distributed systems is to allow sharing of resources. In many cases, sharing resources is done in a cooperative way, as in the case of communication. However, there are also many examples of competitive sharing of resources. For example, two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database. In such cases, it is important that each user does not notice that the other is making use of the same resource. This phenomenon is called **concurrency transparency**. An important issue is that concurrent access to a shared resource leaves that resource in a consistent state. Consistency can be achieved through locking mechanisms, by which users are, in turn, given exclusive access to the desired resource. A more refined mechanism is to make use

of transctions, but as we shall see in later chapters, transactions are difficult to implement in distributed systems.

A popular alternative definition of a distributed system, due to Leslie Lamport, is "You know you have one when the crash of a computer you've never heard of stops you from getting any work done." This description puts the finger on another important issue of distributed systems design: dealing with failures. Making a distributed system **failure transparent** means that a user does not notice that a resource (he has possibly never heard of) fails to work properly, and that the system subsequently recovers from that failure. Masking failures is one of the hardest issues in distributed systems and is even impossible when certain apparently realistic assumptions are made, as we will discuss in Chap. 7. The main difficulty in masking failures lies in the inability to distinguish between a dead resource and a painfully slow resource. For example, when contacting a busy Web server, a browser will eventually time out and report that the Web page is unavailable. At that point, the user cannot conclude that the server is really down.

The last type of transparency that is often associated with distributed systems is **persistence transparency**, which deals with masking whether a resource is in volatile memory or perhaps somewhere on a disk. For example, many object-oriented databases provide facilities for directly invoking methods on stored objects. What happens behind the scenes, is that the database server first copies the object's state from disk to main memory, performs the operation, and perhaps writes that state back to secondary storage. The user, however, is unaware that the server is moving state between primary and secondary memory. Persistence plays an important role in distributed systems, but it is equally important for nondistributed systems.

### Degree of Transparency

Although distribution transparency is generally preferable for any distributed system, there are situations in which attempting to blindly hide all distribution aspects from users is not always a good idea. An example is requesting your electronic newspaper to appear in your mailbox before 7 A.M. local time, as usual, while you are currently at the other end of the world living in a different time zone. Your morning paper will not be the morning paper you are used to.

Likewise, a wide-area distributed system that connects a process in San Francisco to a process in Amsterdam cannot be expected to hide the fact that Mother Nature will not allow it to send a message from one process to the other in less than approximately 35 milliseconds. Practice shows that it actually takes several hundreds of milliseconds using a computer network. Signal transmission is not only limited by the speed of light, but also by limited processing capacities of the intermediate switches.

There is also a trade-off between a high degree of transparency and the performance of a system. For example, many Internet applications repeatedly try to

contact a server before finally giving up. Consequently, attempting to mask a transient server failure before trying another one may slow down the system as a whole. In such a case, it may have been better to give up earlier, or at least let the user cancel the attempts to make contact.

Another example is where we need to guarantee that several replicas, located on different continents, need to be consistent all the time. In other words, if one copy is changed, that change should be propagated to all copies before allowing any other operation. It is clear that a single update operation may now even take seconds to complete, something that cannot be hidden from users.

The conclusion is that aiming for distribution transparency is a nice goal when designing and implementing distributed systems, but that it should be considered together with other issues such as performance.

### 1.2.3 Openness

Another important goal of distributed systems is openness. An **open distributed system** is a system that offers services according to standard rules that describe the syntax and semantics of those services. For example, in computer networks, standard rules govern the format, contents, and meaning of messages sent and received. Such rules are formalized in protocols. In distributed systems, services are generally specified through **interfaces**, which are often described in an **Interface Definition Language (IDL)**. Interface definitions written in an IDL nearly always capture only the syntax of services. In other words, they specify precisely the names of the functions that are available together with types of the parameters, return values, possible exceptions that can be raised, and so on. The hard part is specifying precisely what those services do, that is, the semantics of interfaces. In practice, such specifications are given simply in an informal way by means of natural language.

If properly specified, an interface definition allows an arbitrary process that needs a certain interface to talk to another process that provides that interface. It also allows two independent parties to build completely different implementations of those interfaces, leading to two separate distributed systems that operate in exactly the same way. Proper specifications are complete and neutral. Complete means that everything that is necessary to make an implementation has indeed been specified. However, many interface definitions are not at all complete, so that it is necessary for a developer to add implementation-specific details. Just as important is the fact that specifications do not prescribe what an implementation should look like; they should be neutral. Completeness and neutrality are important for interoperability and portability (Blair and Stefani, 1998). **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 flexible, meaning 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. For example, in a flexible system, it should be relatively easy to add parts that run on a different operating system, or even to replace an entire file system. As many of us know from daily practice, attaining flexibility is easier said than done.

## 1.2.4 Scalability

Worldwide connectivity through the Internet is rapidly becoming at least as common as being able to send a postcard to anyone anywhere around the world. With this in mind, scalability is one of the most important design goals for developers of distributed systems.

Scalability of a system can be measured along at least three different dimensions (Neuman, 1994). First, a system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system. Second, a geographically scalable system is one in which the users and resources may lie far apart. Third, a system can be administratively scalable, meaning that it can still be easy to manage even if it spans many independent administrative organizations. Unfortunately, a system that is scalable in one or more of these dimensions often exhibits some loss of performance as the system scales up.

### Scalability Problems

When a system needs to scale, very different types of problems need to be solved. Let us first consider scaling with respect to size. If more users or resources need to be supported, we are often confronted with the limitations of centralized services, data, and algorithms (see Fig. 1-3). For example, many services are centralized in the sense that they are implemented by means of only a single server running on a specific machine in the distributed system. The problem with this scheme is obvious: the server can simply become a bottleneck as the number of users grows. Even if we have virtually unlimited processing and storage capacity, communication with that server will eventually prohibit further growth.

Unfortunately, using only a single server is sometimes unavoidable. Imagine that we have a service for managing highly confidential information such as medical records, bank accounts, personal loans, and so on. In such cases, it may be best to implement that service by means of a single server in a highly secured separate room, and protected from other parts of the distributed system through special network components. Copying the server to several locations to enhance performance may be out of the question as it would make the service more vulnerable to security attacks.

| Concept | Example |
|---|---|
| Centralized services | A single server for all users |
| Centralized data | A single on-line telephone book |
| Centralized algorithms | Doing routing based on complete information |

**Figure 1-3.** Examples of scalability limitations.

Just as bad as centralized services are centralized data. How should we keep track of the telephone numbers and addresses of 50 million people? Suppose that

each data record could be fit into 50 characters. A single 2.5-gigabyte disk would provide enough storage. But here again, having a single database would undoubtedly saturate all the communication lines into and out of it. Likewise, imagine how the Internet would work if its Domain Name System (DNS) was still implemented as a single table. DNS maintains information on millions of computers worldwide and forms an essential service for locating Web servers. If each request to resolve a URL had to be forwarded to that one and only DNS server, it is clear that no one would be using the Web (which, by the way, would probably solve the problem again).

Finally, centralized algorithms are also a bad idea. In a large distributed system, an enormous number of messages have to be routed over many lines. From a theoretical point of view, the optimal way to do this is collect complete information about the load on all machines and lines, and then run a graph theory algorithm to compute all the optimal routes. This information can then be spread around the system to improve the routing.

The trouble is that collecting and transporting all the input and output information would again be a bad idea because these messages would overload part of the network. In fact, any algorithm that operates by collecting information from all sites, sends it to a single machine for processing, and then distributes the results must be avoided. Only decentralized algorithms should be used. These algorithms generally have the following characteristics, which distinguish them from centralized algorithms:

1. No machine has complete information about the system state.
2. Machines make decisions based only on local information.
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists.

The first three follow from what we have said so far. The last is perhaps less obvious but also important. Any algorithm that starts out with: "At precisely 12:00:00 all machines shall note the size of their output queue" will fail because it is impossible to get all the clocks exactly synchronized. Algorithms should take into account the lack of exact clock synchronization. The larger the system, the larger the uncertainty. On a single LAN, with considerable effort it may be possible to get all clocks synchronized down to a few milliseconds, but doing this nationally or internationally is tricky.

Geographical scalability has its own problems. One of the main reasons why it is currently hard to scale existing distributed systems that were designed for local-area networks is that they are based on **synchronous communication**. In this form of communication, a party requesting service, generally referred to as a client, blocks until a reply is sent back. This approach generally works fine in LANs where communication between two machines is generally at worst a few

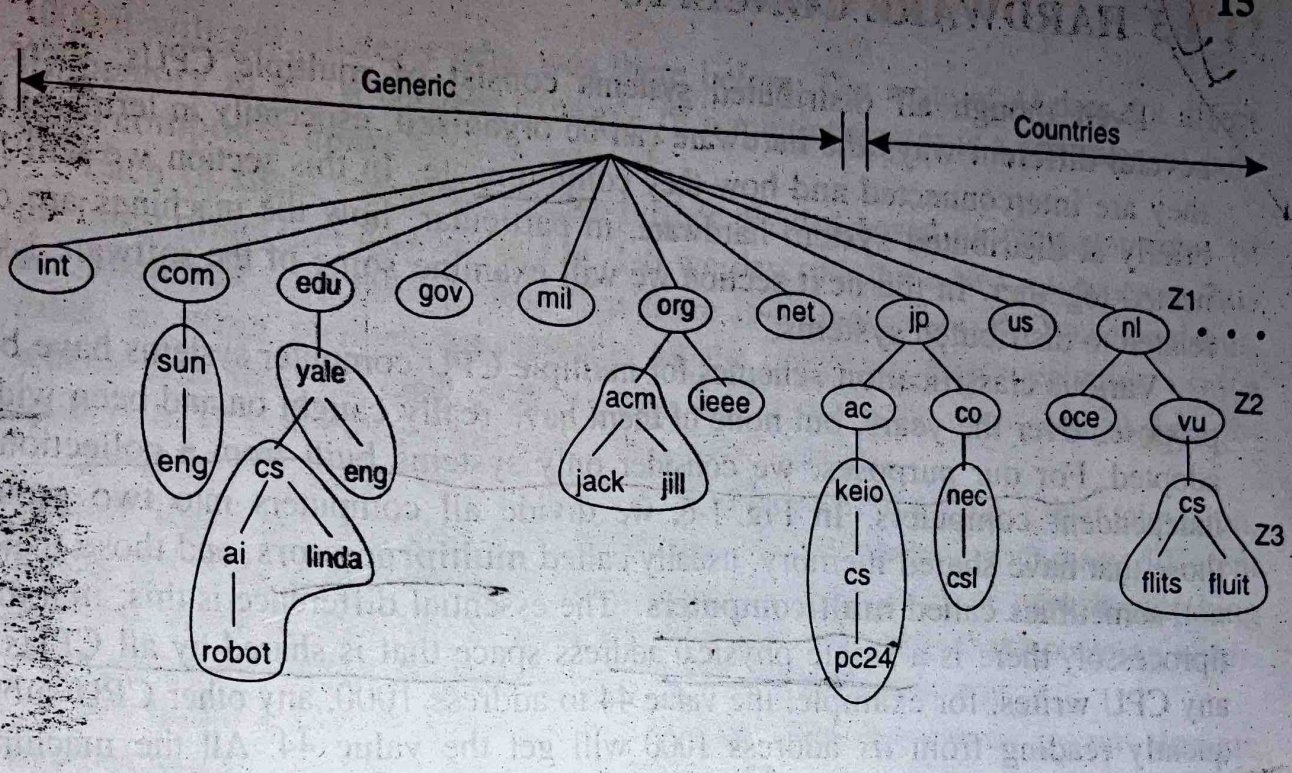


**Figure 1-5.** An example of dividing the DNS name space into zones.

distributed system. Replication not only increases availability, but also helps to balance the load between components leading to better performance. Also, in geographically widely dispersed systems, having a copy nearby can hide much of the communication latency problems mentioned before.

**Caching** is a special form of replication, although the distinction between the two is often hard to make or even artificial. As in the case of replication, caching results in making a copy of a resource, generally in the proximity of the client accessing that resource. However, in contrast to replication, caching is a decision made by the client of a resource, and not by the owner of a resource.

There is one serious drawback to caching and replication that may adversely affect scalability. Because we now have multiple copies of a resource, modifying one copy makes that copy different from the others. Consequently, caching and replication leads to **consistency** problems.

To what extent inconsistencies can be tolerated depends highly on the usage of a resource. For example, many Web users find it acceptable that their browser

## 1.3 HARDWARE CONCEPTS

Even though all distributed systems consist of multiple CPUs, there are several different ways the hardware can be organized, especially in terms of how they are interconnected and how they communicate. In this section we will look briefly at distributed system hardware, in particular, how the machines are connected together. In the next section we will examine some of the software issues related to distributed systems.

Various classification schemes for multiple CPU computer systems have been proposed over the years, but none of them have really caught on and been widely adopted. For our purposes, we consider only systems built from a collection of independent computers. In Fig. 1-6, we divide all computers into two groups: those that have shared memory, usually called **multiprocessors**, and those that do not, sometimes called **multicomputers**. The essential difference is this: in a multiprocessor, there is a single physical address space that is shared by all CPUs. If any CPU writes, for example, the value 44 to address 1000, any other CPU subsequently reading from *its* address 1000 will get the value 44. All the machines share the same memory.

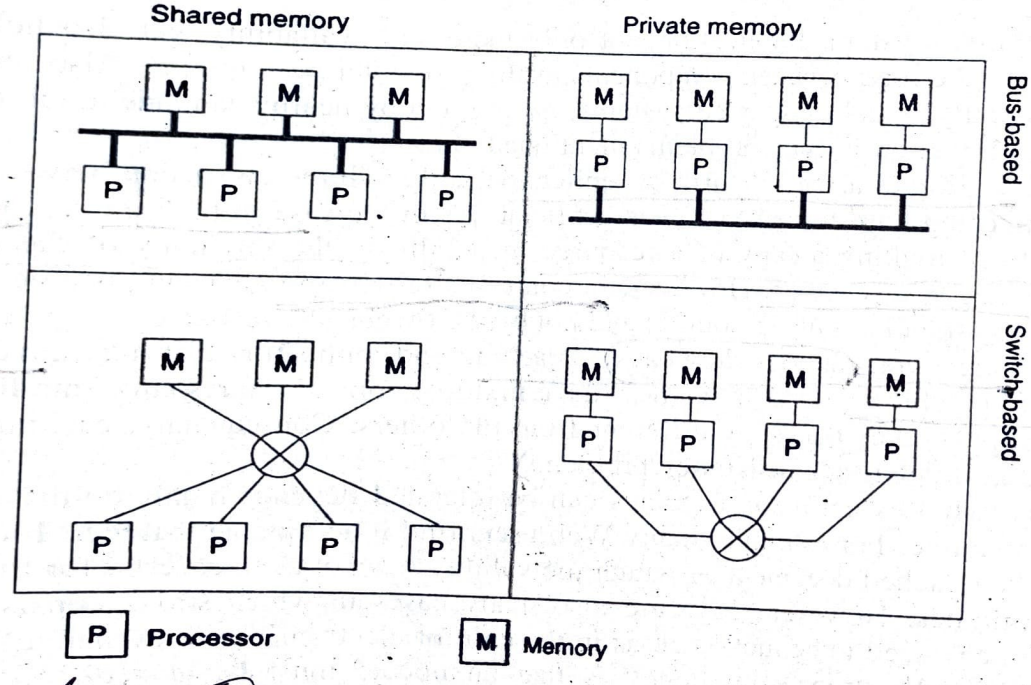


**Figure 1-6.** Different basic organizations of processors and memories in distributed computer systems.

In contrast, in a multicomputer, every machine has its own private memory. After one CPU writes the value 44 to address 1000, if another CPU reads address

1000 it will get whatever value was there before. The write of 44 does not affect *its* memory at all. A common example of a multicomputer is a collection of personal computers connected by a network.

Each of these categories can be further divided based on the architecture of the interconnection network. In Fig. 1-6 we describe these two categories as **bus** and **switched**. By bus we mean that there is a single network, backplane, bus, cable, or other medium that connects all the machines. Cable television uses a scheme like this: the cable company runs a wire down the street, and all the subscribers have taps running to it from their television sets.

Switched systems do not have a single backbone like cable television. Instead, there are individual wires from machine to machine with many different wiring patterns in use. Messages move along the wires, with an explicit switching decision made at each step to route the message along one of the outgoing wires. The worldwide public telephone system is organized in this way.

We make a further distinction between distributed computer systems that are **homogeneous** and those that are **heterogeneous**. This distinction is useful only for multicomputers. In a homogeneous multicomputer, there is essentially only a single interconnection network that uses the same technology everywhere. Likewise, all processors are the same and generally have access to the same amount of private memory. Homogeneous multicomputers tend to be used more as parallel systems (working on a single problem), just like multiprocessors.

In contrast, a heterogeneous multicomputer system may contain a variety of different, independent computers, which in turn are connected through different networks. For example, a distributed computer system may be constructed from a collection of different local-area computer networks, which are interconnected through an FDDI or ATM-switched backbone.

In the following three sections, we will take a closer look at multiprocessors, and homogeneous and heterogeneous multicomputer systems. Although these topics are not directly related to our main concern, distributed systems, they will shed some light on the subject because the organization of distributed systems often depends on the underlying hardware.

## 1.3.1 Multiprocessors

## 1.4 SOFTWARE CONCEPTS

Hardware for distributed systems is important, but it is software that largely determines what a distributed system actually looks like. Distributed systems are very much like traditional operating systems. First, they act as **resource managers** for the underlying hardware, allowing multiple users and applications to share resources such as CPUs, memories, peripheral devices, the network, and data of all kinds. Second, and perhaps more important, is that distributed systems attempt to hide the intricacies and heterogeneous nature of the underlying hardware by providing a virtual machine on which applications can be easily executed.

To understand the nature of distributed systems, we will therefore first take a look at operating systems in relation to distributed computers. Operating systems for distributed computers can be roughly divided into two categories: tightly-coupled systems and loosely-coupled systems. In tightly-coupled systems, the operating system essentially tries to maintain a single, global view of the resources it manages. Loosely-coupled systems can be thought of as a collection of computers each running their own operating system. However, these operating systems work together to make their own services and resources available to the others.

This distinction between tightly-coupled and loosely-coupled systems is related to the hardware classification given in the previous section. A tightly-coupled operating system is generally referred to as a **distributed operating system (DOS)**, and is used for managing multiprocessors and homogeneous multicomputers. Like traditional uniprocessor operating systems, the main goal of a distributed operating system is to hide the intricacies of managing the underlying hardware such that it can be shared by multiple processes.

The loosely-coupled **network operating system (NOS)** is used for heterogeneous multicomputer systems. Although managing the underlying hardware is an important issue for a NOS, the distinction from traditional operating systems comes from the fact local services are made available to remote clients. In the following sections we will first take a look at tightly-coupled and loosely-coupled operating systems.

To actually come to a distributed system, enhancements to the services of network operating systems are needed such that a better support for distribution transparency is provided. These enhancements lead to what is known as **middleware**, and lie at the heart of modern distributed systems. Middleware is also discussed in this section Fig. 1-10 summarizes the main issues with respect to DOS, NOS, and middleware.

### 1.4.1 Distributed Operating Systems

There are two types of distributed operating systems. A **multiprocessor operating system** manages the resources of a multiprocessor. A **multicomputer operating system** is an operating system that is developed

| System | Description | Main goal |
|---|---|---|
| DOS | Tightly-coupled operating system for multi-processors and homogeneous multicomputers | Hide and manage hardware resources |
| NOS | Loosely-coupled operating system for hetero-geneous multicomputers (LAN and WAN) | Offer local services to remote clients |
| Middleware | Additional layer atop of NOS implementing general-purpose services | Provide distribution transparency |

**Figure 1-10.** An overview between DOS (Distributed Operating Systems). NOS (Network Operating Systems), and middleware.

multicomputers. The functionality of distributed operating systems is essentially the same as that of traditional operating systems for uniprocessor systems, except that they handle multiple CPUs. Let us therefore briefly review uniprocessor operating systems first. An introduction to operating systems for uniprocessors and multiple processors can be found in (Tanenbaum, 2001).

### Uniprocessor Operating Systems

Operating systems have traditionally been built to manage computers with only a single CPU. The main goal of these systems is to allow users and applications an easy way of sharing resources such as the CPU, main memory, disks, and peripheral devices. Sharing resources means that different applications can make use of the same hardware in an isolated fashion. To an application, it appears as if it has its own resources, and that there may be several applications executing on the same system at the same time, each with their own set of resources. In this sense, the operating system is said to implement a **virtual machine**, offering multitasking facilities to applications.

An important aspect of sharing resources in such a virtual machine, is that applications are protected from each other. For example, it is not acceptable that if two independent applications *A* and *B* are executed at the same time, that *A* can alter the data of application *B* by simply accessing that part of main memory where that data are currently stored. Likewise, we need to ensure that applications can make use of facilities only as offered by the operating system. For instance, it should generally be prevented that an application can directly copy messages to a network interface. Instead, the operating system will provide communication primitives, and only by means of these primitives should it be possible to send messages between applications on different machines.

Consequently, the operating system should be in full control of how the hardware resources are used and shared. Therefore, most CPUs support at least two modes of operation. In **kernel mode**, all instructions are permitted to be executed, and the whole memory and collection of all registers is accessible during

## 1.4.3 Middleware

Neither a distributed operating system or a network operating system really qualifies as a distributed system according to our definition given in Sec. 1.1. A distributed operating system is not intended to handle a collection of *independent* computers, while a network operating system does not provide a view of a *single coherent system*. The question comes to mind whether it is possible to develop a distributed system that has the best of both worlds: the scalability and openness of network operating systems and the transparency and related ease of use of distributed operating systems. The solution is to be found in an additional layer of software that is used in network operating systems to more or less hide the heterogeneity of the collection of underlying platforms but also to improve distribution transparency. Many modern distributed systems are constructed by means of such an additional layer of what is called **middleware**. In this section we take a closer look at what middleware actually constitutes by explaining some of its features.

### Positioning Middleware

Many distributed applications make direct use of the programming interface offered by network operating systems. For example, communication is often expressed through operations on sockets, which allow processes on different machines to pass each other messages (Stevens, 1998). In addition, applications often make use of interfaces to the local file system. As we explained, a problem with this approach is that distribution is hardly transparent. A solution is to place an additional layer of software between applications and the network operating system, offering a higher level of abstraction. Such a layer is accordingly called **middleware**. It sits in the middle between applications and the network operating system as shown in Fig. 1-22.

Each local system forming part of the underlying network operating system is assumed to provide local resource management in addition to simple communication means to connect to other computers. In other words, middleware itself will not manage an individual node; this is left entirely to the local operating system.

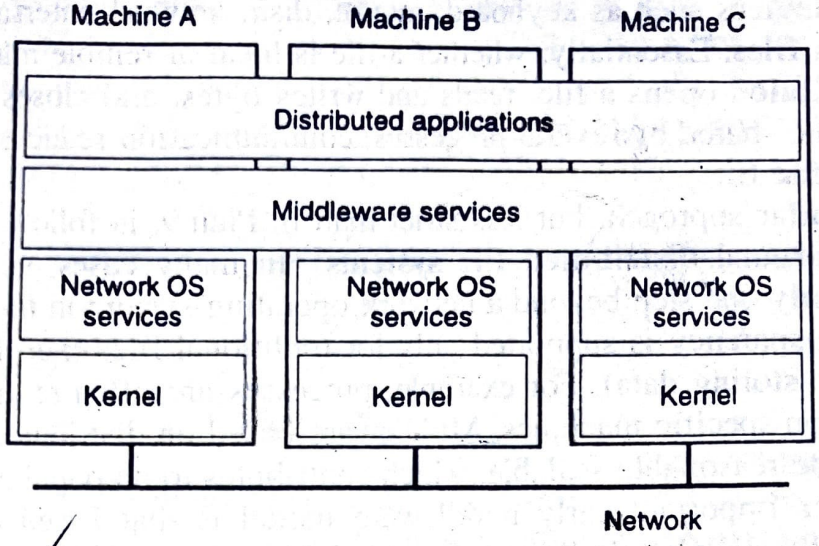


**Figure 1-22.** General structure of a distributed system as middleware.

An important goal is to hide heterogeneity of the underlying platforms from applications. Therefore, many middleware systems offer a more-or-less complete collection of services and discourage using anything else but their interfaces to those services. In other words, skipping the middleware layer and immediately calling services of one of the underlying operating systems is often frowned upon. We will return to middleware services shortly.

It is interesting to note that middleware was not invented as an academic exercise in achieving distribution transparency. After the introduction and widespread use of network operating systems, many organizations found themselves having lots of networked applications that could not be easily integrated into a single system (Bernstein, 1996). At that point, manufacturers started to build higher-level, application-independent services into their systems. Typical examples include support for distributed transactions and advanced communication facilities.

Of course, agreeing on what the right middleware should be is not easy. An approach is to set up an organization which subsequently defines a common standard for some middleware solution. At present, there are a number of such standards available. The standards are generally not compatible with each other, and even worse, products implementing the same standard but from different manufacturers rarely interwork. Surely, it will not be long before someone offers "upperware" to remedy this defect.

## Middleware Models

To make development and integration of distributed applications as simple as possible, most middleware is based on some model, or *paradigm*, for describing distribution and communication. A relatively simple model is that of treating everything as a file. This is the approach originally introduced in UNIX and

offer to applications.

## A Comparison between Systems

A brief comparison between distributed operating systems, network operating systems, and (middleware-based) distributed systems is given in Fig. 1-24.

| Item | Distributed OS | | Network OS | Middleware-based DS |
|---|---|---|---|---|
| | Multiproc. | Multicomp. | | |
| Degree of transparency | Very high | High | Low | High |
| Same OS on all nodes? | Yes | Yes | No | No |
| Number of copies of OS | 1 | N | N | N |
| Basis for communication | Shared memory | Messages | Files | Model specific |
| Resource management | Global, central | Global, distributed | Per node | Per node |
| Scalability | No | Moderately | Yes | Varies |
| Openness | Closed | Closed | Open | Open |

**Figure 1-24.** A comparison between multiprocessor operating systems, multicomputer operating systems, network operating systems, and middleware-based distributed systems.

With respect to transparency, it is clear that distributed operating systems do a better job than network operating systems. In multiprocessor systems we have to hide only that there are more processors, which is relatively easy. The hard part is also hiding that memory is physically distributed, which is why building multi-

multiprocessor systems (i.e., [illegible]
systems are often hard to scale.

Finally, network operating systems and distributed systems comes to openness. In general, nodes support a standard communication protocol such as TCP/IP, making interoperability easy. However, there may be a lot of problems porting applications when many different kinds of operating systems are used. In general, distributed operating systems are not designed to be open. Instead, they are often optimized for performance, leading to many proprietary solutions that stand in the way of an open system.

M.T.

## 1.5 THE CLIENT-SERVER MODEL

Up to this point, we have hardly said anything on the actual organization of distributed systems, which mainly centers around the question of how to organize the *processes* in a system. Despite that, consensus on many distributed systems issues is often hard to find, there is one issue that many researchers and practitioners agree upon: thinking in terms of *clients* that request services from *servers* helps understanding and managing the complexity of distributed systems. In this section, we take a closer look at the client-server model.

### 1.5.1 Clients and Servers

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 Fig. 1-25

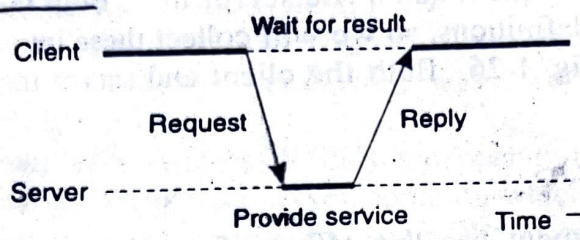


**Figure 1-25.** General interaction between a client and a server.

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. It is not hard to see that there is no single solution to this problem. We defer a detailed discussion on handling transmission failures to Chap. 7.

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 is that setting up and tearing down a connection is relatively costly, especially when the request and reply messages are small. We will discuss an alternative solution where connection management is combined with data transfer in the next chapter.

shown, and no error checking is done. How the client and server interact should be clear. In the following sections, we discuss some more of the organizational issues of the client-server model.

### 1.5.2 Application Layering

The client-server model has been subject to many debates and controversies. 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 essentially does no more than process queries.

However, considering that many client-server applications are targeted toward supporting user access to databases, many people have advocated a distinction between the following three levels:

1. The user-interface level
2. The processing level
3. The data level

The user-interface level contains all that is necessary to directly interface with the user, such as display management. The processing level typically contains the applications, whereas the data level contains the actual data that is being acted on. In the following sections, we discuss each of these levels.

#### User-Interface Level

Clients typically implement the user-interface level. This level consists of the programs that allow end users to interact with applications. There is a considerable difference in how sophisticated user-interface programs are.

The simplest user-interface program is nothing more than a character-based screen. Such an interface has been typically used in mainframe environments. In those cases where the mainframe controls all interaction, including the keyboard and monitor, one can hardly speak of a client-server environment. However, in

many cases, the user's terminal does some local processing such as echoing typed keystrokes, or supporting form-like interfaces in which a complete entry is to be edited before sending it to the main computer.

Nowadays, even in mainframe environments, we see more advanced user interfaces. Typically, the client machine offers at least a graphical display in which pop-up or pull-down menus are used, and of which much of the screen controls are handled through a mouse instead of the keyboard. Typical examples of such interfaces include the X-Windows interfaces as used in many UNIX environments, and earlier interfaces developed for MS-DOS PCs and Apple Macintoshes.

Modern user interfaces offer considerably more functionality by allowing applications to share a single graphical window, and to use that window to exchange data through user actions. For example, to delete a file, it is often possible to move the icon representing that file to an icon representing a trash can. Likewise, many word processors allow a user to move text in a document to another position by using only the mouse. We return to user interfaces in Chap. 3.

### Processing Level

Many client-server applications can be constructed from roughly three different pieces: a part that handles interaction with a user, a part that operates on a database or file system, and a middle part that generally contains the core functionality of an application. This middle part is logically placed at the processing level. In contrast to user interfaces and databases, there are not many aspects common to the processing level. Therefore, we shall give a number of examples to make this level clearer.

As a first example, consider an Internet search engine. Ignoring all the animated banners, images, and other fancy window dressing, the user interface of a search engine is very simple: a user types in a string of keywords and is subsequently presented with a list of titles of Web pages. The back end is formed by a huge database of Web pages that have been prefetched and indexed. The core of the search engine is a program that transforms the user's string of keywords into one or more database queries. It subsequently ranks the results into a list, and transforms that list into a series of HTML pages. Within the client-server model, this information retrieval part is typically placed at the processing level. Fig. 1-28 shows this organization.

As a second example, consider a decision support system for a stock brokerage. Analogous to a search engine, such a system can be divided into a front end implementing the user interface, a back end for accessing a database with the financial data, and the analysis programs between these two. Analysis of financial data may require sophisticated methods and techniques from statistics and artificial intelligence. In some cases, the core of a financial decision support system may even need to be executed on high-performance computers in order to achieve the throughput and responsiveness that is expected from its users.

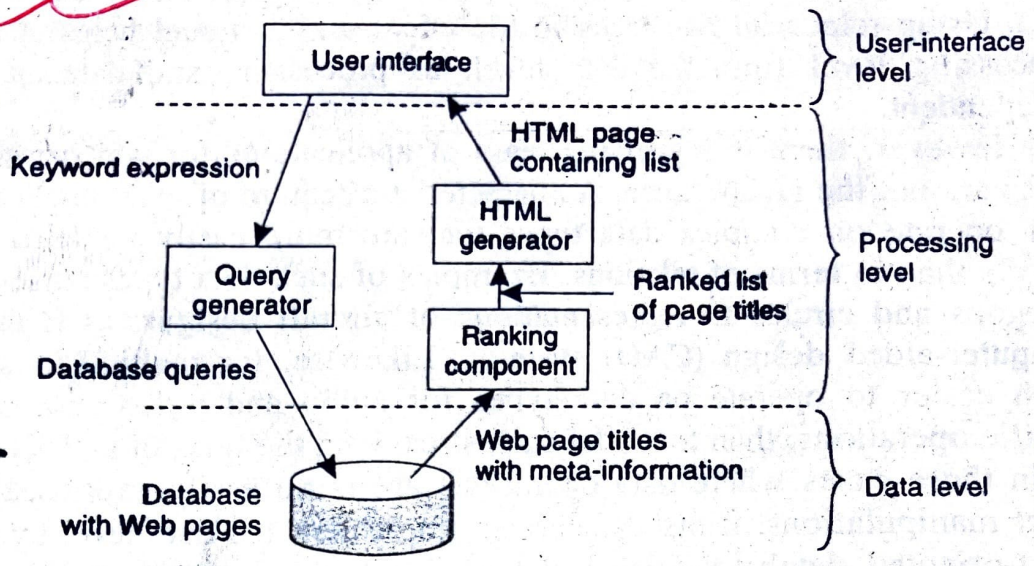


**Figure 1-28.** The general organization of an Internet search engine into three different layers.

As a last example, consider a typical desktop package, consisting of a word processor, a spreadsheet application, communication facilities, and so on. Such "office" suites are generally integrated through a common user interface that supports compound documents, and operates on files from the user's home directory. In this case, the processing level consists of a relatively large collection of programs, each having rather simple processing capabilities.

### Data Level

The data level in the client-server model contains the programs that maintain the actual data on which the applications operate. An important property of this level is that data are persistent, that is, even if no application is running, data will be stored somewhere for next use. In its simplest form, the data level consists of a file system, but it is more common to use a full-fledged database. In the client-server model, the data level is typically implemented at the server side.

Besides merely storing data, the data level is generally also responsible for keeping data consistent across different applications. When databases are being used, maintaining consistency means that metadata such as table descriptions, entry constraints and application-specific metadata are also stored at this level. For example, in the case of a bank, we may want to generate a notification when a customer's credit card debt reaches a certain value. This type of information can be maintained through a database trigger that activates a handler for that trigger at the appropriate moment.

In traditional business-oriented environments, the data level is organized as a relational database. Data independence is a keyword here. The data are organized independent of the applications in such a way that changes in that organization do

not affect applications, and neither do the applications affect the data organization. Using relational databases in the client-server model helps us separate the processing level from the data level, as processing and data are considered independent.

However, there is a growing class of applications for which relational databases are not the ideal choice. A characteristic feature of these applications is that they operate on complex data types that are more easily modeled in terms of objects than in terms of relations. Examples of such data types range from simple polygons and circles to representations of aircraft designs, as is the case with computer-aided design (CAD) systems. Likewise, for multimedia systems it is much easier to operate on data types for audio and video streams with their specific operations, than to model such streams in the form of tables of relations.

In those cases where data operations are more easily expressed in terms of object manipulations, it makes sense to implement the data level by means of an object-oriented database. Such a database not only supports the organization of complex data in terms of objects, but also stores the implementation of the operations on those objects. Consequently, part of the functionality that was found in the processing level is now moved to the data level.

### 1.5.3 Client-Server Architectures

The distinction into three logical levels as discussed in the previous section, 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

The problem with this organization is that it is not really distributed: everything is handled by the server, while the client is essentially no more than a dumb terminal. There are many other possibilities, of which we explore some of the more common ones in this section.

#### Multitiered Architectures

One approach for organizing clients and servers is to distribute the programs in the application layers of the previous section across different machines, as shown in Fig. 1-29 (see also Umar, 1997; Jing et al., 1999). As a first step, we

make a distinction between only two kinds of machines: clients and servers, leading to what is also referred to as a **(physically) two-tiered architecture**.

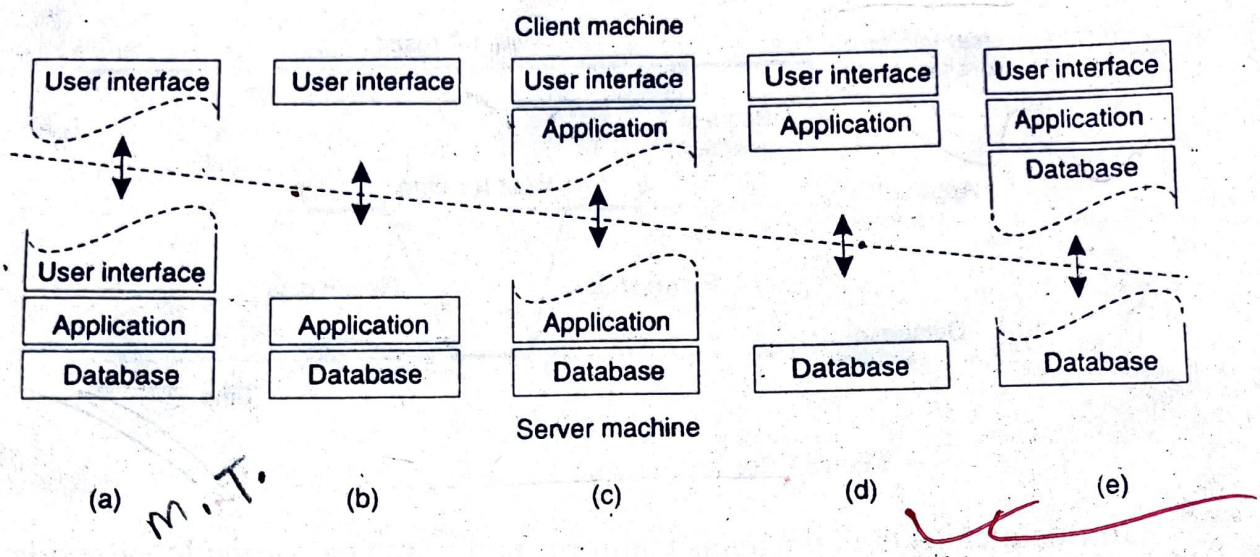


**Figure 1-29.** Alternative client-server organizations (a)-(e).

One possible organization is to have only the terminal-dependent part of the user interface on the client machine, as shown in Fig. 1-29(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 Fig. 1-29(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 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 Fig. 1-29(c). An example where this makes sense is where the application makes use of a form that needs to be filled in entirely before it can be processed. The front end can then check the correctness and consistency of the form, and where necessary interact with the user. Another example of the organization of Fig. 1-29(c), is that of a word processor in which the basic editing functions execute on the client side where they operate on locally cached, or in-memory data, but where the advanced support tools such as checking the spelling and grammar execute on the server side.

In many client-server environments, the organizations shown in Fig. 1-29(d) and Fig. 1-29(e) are particular popular. These organizations are used in the case 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. Fig. 1-29(e) represents the situation where the client's local disk contains part of the data. For example, when browsing the Web, a client can gradually build a huge cache on local disk of most recent inspected Web pages.

When distinguishing only clients and servers, we miss the point that a server may sometimes need to act as a client, as shown in Fig. 1-30, leading to a (physically) three-tiered architecture.

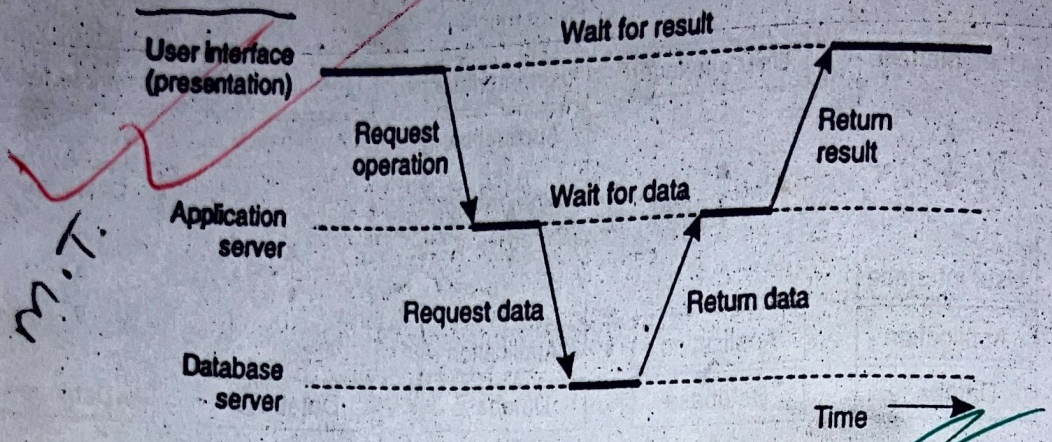


Figure 1-30. An example of a server acting as client.

In this architecture, programs that form part of the processing level reside on a separate server, but may additionally be partly distributed across the client and server machines. A typical example of where a three-tiered architecture is used is in transaction processing. In this case, a separate process, called the transaction monitor, coordinates all transactions across possibly different data servers. We return to transaction processing in later chapters.

## Modern Architectures

Multitiered client-server architectures are a direct consequence of dividing applications into a user-interface, processing components, and a data level. The different tiers correspond directly with the logical organization of applications. In many business environments, distributed processing is equivalent to organizing a client-server application as a multitiered architecture. We refer to this type of distribution as **vertical distribution**. The characteristic feature of vertical distribution is that it is achieved by placing logically different components on different machines. The term is related to the concept of *vertical fragmentation* as used in distributed relational databases, where it means that tables are split column-wise, and subsequently distributed across multiple machines (Ozsu and Valduriez, 1999).

However, vertical distribution is only one way of organizing client-server applications, and in many cases the least interesting one. In modern architectures, it is often the distribution of the clients and the servers that counts, which we refer to as **horizontal distribution**. In this type of distribution, a client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set, thus balancing the load.

As an example of a popular horizontal distribution, consider a Web server replicated across several machines in a local-area network, as shown in Fig. 1-31. Each server has the same set of Web pages, and each time a Web page is updated, a copy is immediately placed at each server. When a request comes in, it is forwarded to a server using a round-robin policy. It turns out that this form of horizontal distribution can be quite effective for highly popular Web sites, provided enough bandwidth is available.

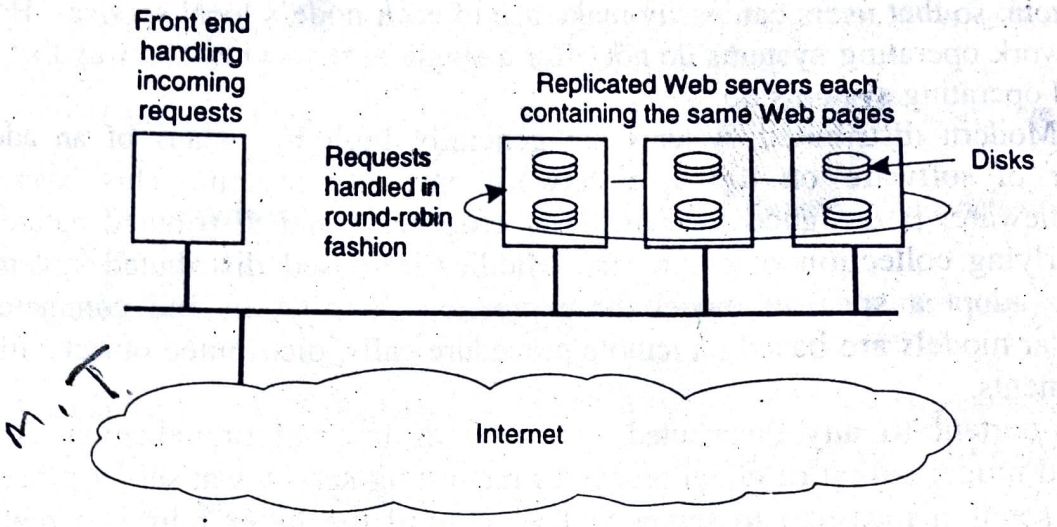


**Figure 1-31.** An example of horizontal distribution of a Web service.

Although less apparent, clients can be distributed as well. For simple collaborative applications, we may even have the case where there is no server at all. In such a case, we often talk about **peer-to-peer distribution**. What may happen, for example, is that a user seeks contact with another user, after which both launch the same application for starting a session. A third client may contact either one of the two, and subsequently also launch the same application software.

A number of alternative organizations for client-server systems are discussed in (Adler, 1995). We will come across many other organizations for distributed systems as well in later chapters. We will see that systems are generally distributed both in a vertical and horizontal sense.

## 1.6 SUMMARY

Distributed systems consist of autonomous computers that work together to give the appearance of a single coherent system. One important advantage is that they make it easier to integrate different applications running on different computers into a single system. Another advantage is that when properly designed, distributed systems scale well with respect to the size of the underlying network. These advantages often come at the cost of more complex software, degradation

of performance, and also often weaker security. Nevertheless, there is considerable interest worldwide in building and installing distributed systems.

Different types of distributed systems exist. A distributed operating system distinguishes itself by managing the hardware of tightly-coupled computer systems, which include multiprocessors and homogeneous multicomputers. These distributed systems do not really support autonomous computers, but do a good job at providing a single-system view. A network operating system, on the other hand, is good at connecting different computers, each with their own operating system, so that users can easily make use of each node's local services. However, network operating systems do not offer a single-system view the way that distributed operating systems do.

Modern distributed systems are generally built by means of an additional layer of software on top of a network operating system. This layer, called middleware, is designed to hide the heterogeneity and distributed nature of the underlying collection of computers. Middleware-based distributed systems generally adopt a specific model for expressing distribution and communication. Popular models are based on remote procedure calls, distributed objects, files, and documents.

Important to any distributed system is its internal organization. A widely applied model is that of client processes requesting services at server processes. A client sends a message to server and waits until the latter returns a reply. This model is strongly related to traditional programming, in which services are implemented as procedures in separate modules. A further refinement is often made by distinguishing a user-interface level, a processing level, and a data level. The server is generally responsible for the data level, whereas the user-interface level is implemented at the client side. The processing level can be implemented at the client, the server, or split between the two.

For modern distributed systems, this vertical organization of client-server applications is not sufficient to build large-scale systems. What is needed is a horizontal distribution by which clients and servers are physically distributed and replicated across multiple computers. A typical example in which horizontal distribution has been successfully applied is the World Wide Web.

## PROBLEMS

1. What is the role of middleware in a distributed system?
2. Explain what is meant by (distribution) transparency, and give examples of different types of transparency.
3. Why is it sometimes so hard to hide the occurrence and recovery from failures in a distributed system?