INTRODUCTION

Computer systems are undergoing a revolution. From 1945, when the modern computer era began, until about 1985, computers were large and expensive. Even minicomputers cost tens of thousands of dollars each. As a result, most organizations had only a handful of computers, and for lack of a way to connect them, these 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. Many of these had the computing power of a mainframe (i.e., large) computer, but for a fraction of the price.

The amount of improvement that has occurred in computer technology in the past half century is truly staggering and totally unprecedented in other industries. From a machine that cost 100 million dollars and executed 1 instruction per second, we have come to machines that cost 1000 dollars and are able to execute 10 million instructions per second, a price/performance gain of $10^{12}$. If cars had improved at this rate in the same time period, a Rolls Royce would now cost 1 dollar and get a billion miles per gallon. (Unfortunately, it would probably also have a 200-page manual telling how to open the door.)

The second development was the invention of high-speed computer networks. **Local-area networks** or **LANs** allow hundreds of machines within a building to be connected in such a way that small amounts of information can be transferred between machines in a few microseconds or so. Larger amounts of data can be
moved between machines at rates of 10 to 1000 million bits/sec. **Wide-area networks** or **WANs** allow millions of machines all over the earth to be connected at speeds varying from 64 Kbps (kilobits per second) to gigabits per second.

The result of these technologies is that it is now not only feasible, but easy, to put together computing systems composed of large numbers of computers connected by a high-speed network. They are usually called computer networks or **distributed systems**, in contrast to the previous **centralized systems** (or **single-processor systems**) consisting of a single computer, its peripherals, and perhaps some remote terminals.

### 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.](image)

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 document, a
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.

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

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 its 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 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 transactions, 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.

### Separating Policy from Mechanism

To achieve flexibility in open distributed systems, it is crucial that the system is organized as a collection of relatively small and easily replaceable or adaptable components. This implies that we should provide definitions of not only the highest-level interfaces, that is, those seen by users and applications, but also definitions for interfaces to internal parts of the system and describe how those parts interact. This approach is relatively new. Many older and even contemporary systems are constructed using a monolithic approach in which components are only logically separated but implemented as one, huge program. This approach makes it hard to replace or adapt a component without affecting the entire system. Monolithic systems thus tend to be closed instead of open.

The need for changing a distributed system is often caused by a component that does not provide the optimal policy for a specific user or application. As an example, consider caching in the World Wide Web. Browsers generally allow a user to adapt their caching policy by specifying the size of the cache, and whether a cached document should always be checked for consistency, or perhaps only once per session. However, the user cannot influence other caching parameters, such as how long a document may remain in the cache, or which document should be removed when the cache fills up. Also, it is impossible to make caching decisions based on the content of a document. For instance, a user may want to cache railroad timetables knowing that these hardly change, but never information on current traffic conditions on the highways.

What we need is a separation between policy and mechanism. In the case of Web caching, for example, a browser should ideally provide facilities for only storing documents, and at the same time allow users to decide which documents are stored and for how long. In practice, this can be implemented by offering a rich set of parameters that the user can set (dynamically). Even better is that a user can implement his own policy in the form of a component that can be plugged into the browser. Of course, that component must have an interface that the browser can understand so that it can procedures of that interface.
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.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized services</td>
<td>A single server for all users</td>
</tr>
<tr>
<td>Centralized data</td>
<td>A single on-line telephone book</td>
</tr>
<tr>
<td>Centralized algorithms</td>
<td>Doing routing based on complete information</td>
</tr>
</tbody>
</table>

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
SEC. 1.2 GOALS

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
hundred microseconds. However, in a wide-area system, we need to take into account that interprocess communication may be hundreds of milliseconds, three orders of magnitude slower. Building interactive applications using synchronous communication in wide-area systems requires a great deal of care (and not a little patience).

Another problem that hinders geographical scalability is that communication in wide-area networks is inherently unreliable, and virtually always point-to-point. In contrast, local-area networks generally provide highly reliable communication facilities based on broadcasting, making it much easier to develop distributed systems. For example, consider the problem of locating a service. In a local-area system, a process can simply broadcast a message to every machine, asking if it is running the service it needs. Only those machines that have that service respond, each providing its network address in the reply message. Such a location scheme is unthinkable in a wide-area system. Instead, special location services need to be designed, which may need to scale worldwide and be capable of servicing a billion users. We return to such services in Chap. 4.

Geographical scalability is strongly related to the problems of centralized solutions that hinder size scalability. If we have a system with many centralized components, it is clear that geographical scalability will be limited due to the performance and reliability problems resulting from wide-area communication. In addition, centralized components now lead to a waste of network resources. Imagine that a single mail server is used for an entire country. This would mean that sending an e-mail to your neighbor would first have to go to the central mail server, which may be hundreds of miles away. Clearly, this is not the way to go.

Finally, a difficult, and in many cases open question is how to scale a distributed system across multiple, independent administrative domains. A major problem that needs to be solved is that of conflicting policies with respect to resource usage (and payment), management, and security.

For example, many components of a distributed system that reside within a single domain can often be trusted by users that operate within that same domain. In such cases, system administration may have tested and certified applications, and may have taken special measures to ensure that such components cannot be tampered with. In essence, the users trust their system administrators. However, this trust does not expand naturally across domain boundaries.

If a distributed system expands to another domain, two types of security measures need to be taken. First, the distributed system has to protect itself against malicious attacks from the new domain. For example, users from the new domain may have only read access to the system's file service in its original domain. Likewise, facilities such as expensive image setters or high-performance computers may not be made available to other users. Second, the new domain has to protect itself against malicious attacks from the distributed system. A typical example is that of downloading programs such as applets in Web browsers. Basically, the new domain does not know what to expect from such foreign code, and
may therefore decide to severely limit the access rights for such code. The problem, as we shall see in Chap. 8, is how to enforce those limitations.

Scaling Techniques

Having discussed some of the scalability problems brings us to the question of how those problems can generally be solved. Because scalability problems in distributed systems appear as performance problems caused by limited capacity of servers and network, there are basically only three techniques for scaling: hiding communication latencies, distribution, and replication (see also Neuman, 1994).

Hiding communication latencies is applicable in the case of geographical scalability. The basic idea is simple: try to avoid waiting for responses to remote service requests as much as possible. For example, when a service has been requested at a remote machine, an alternative to waiting for a reply from the server is to do other useful work at the requester’s side. Essentially, this means constructing the requesting application in such a way that it uses only asynchronous communication. When a reply comes in, the application is interrupted and a special handler is called to complete the previously issued request. Asynchronous communication can often be used in batch-processing systems and parallel applications, in which more or less independent tasks can be scheduled for execution while another task is waiting for communication to complete. Alternatively, a new thread of control can be started to perform the request. Although it blocks waiting for the reply, other threads in the process can continue.

However, there are many applications that cannot make effective use of asynchronous communication. For example, in interactive applications when a user sends a request he will generally have nothing better to do than to wait for the answer. In such cases, a much better solution is to reduce the overall communication, for example, by moving part of the computation that is normally done at the server to the client process requesting the service. A typical case where this approach works is accessing databases using forms. Normally, filling in forms is done by sending a separate message for each field, and waiting for an acknowledgement from the server, as shown in Fig. 1-4(a). For example, the server may check for syntactic errors before accepting an entry. A much better solution is to ship the code for filling in the form, and possibly checking the entries, to the client, and have the client return a completed form, as shown in Fig. 1-4(b). This approach of shipping code is now widely supported by the Web in the form of Java applets.

Another important scaling technique is distribution. Distribution involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system. A good example of distribution is the Internet Domain Name System (DNS). The DNS name space is hierarchically organized into a tree of domains, which are divided into nonoverlapping zones, as shown in Fig. 1-5. The names in each zone are handled by a single name server. Without
going into too many details, one can think of each path name being the name of a host in the Internet, and is thus associated with a network address of that host. Basically, resolving a name means returning the network address of the associated host. Consider, for example, the name \texttt{nl.vu.cs.flits}. To resolve this name, it is first passed to the server of zone \textit{Z1} (see Fig. 1-5) which returns the address of the server for zone \textit{Z2}, to which the rest of name, \texttt{vu.cs.flits}, can be handed. The server for \textit{Z2} will return the address of the server for zone \textit{Z3}, which is capable of handling the last part of the name and will return the address of the associated host.

This examples illustrates how the \textit{naming service}, as provided by DNS, is distributed across several machines, thus avoiding that a single server has to deal with all requests for name resolution.

As another example, consider the World Wide Web. To most users, the Web appears to be an enormous document-based information system in which each document has its own unique name in the form of a URL. Conceptually, it may even appear as if there is only a single server. However, the Web is physically distributed across a large number of servers, each handling a number of Web documents. The name of the server handling a document is encoded into that document’s URL. It is only because of this distribution of documents that the Web has been capable of scaling to its current size.

Considering that scalability problems often appear in the form of performance degradation, it is generally a good idea to actually \textbf{replicate} components across a
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 returns a cached document of which the validity has not been checked for the last few minutes. However, there are also many cases in which strong consistency guarantees need to be met, such as in the case of electronic stock exchanges. The problem with strong consistency is that an update must be immediately propagated to all other copies. Moreover, if two updates happen concurrently, it is often also required that each copy is updated in the same order. Situations such as these generally require some global synchronization mechanism. Unfortunately, such mechanisms are extremely hard or even impossible to implement in a scalable way. Consequently, scaling by replication may introduce other, inherently nonscalable solutions. We return to replication and consistency in Chap. 6.
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.

![Diagram of 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

SEC. 1.3

1000 it will read the value 44 that the particular CPU wrote to its memory, not the value 44 written by another CPU.

Each machine in the interconnection network can be a switch or a bus. A bus is a passive conductor such as wire or cable, and a switch is an active device such as a relay or semiconductor switch.  

Switching systems are connected by a network of switches. Instead, the bus-based systems have wiring points at each decision point in the network. The wiring point serves as

The word...
SEC. 1.3 HARDWARE CONCEPTS

100 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

Multiprocessor systems all share a single key property: all the CPUs have direct access to the shared memory. Bus-based multiprocessors consist of some number of CPUs all connected to a common bus, along with a memory module. A simple configuration is to have a high-speed backplane or motherboard into which CPU and memory cards can be inserted.

Since there is only one memory, if CPU A writes a word to memory and then CPU B reads that word back a microsecond later, B will get the value just written. A memory that has this property is said to be coherent. The problem with this scheme is that with as few as 4 or 5 CPUs, the bus will usually be overloaded and
performance will drop drastically. The solution is to add a high-speed cache memory between the CPU and the bus, as shown in Fig. 1-7. The cache holds the most recently accessed words. All memory requests go through the cache. If the word requested is in the cache, the cache itself responds to the CPU, and no bus request is made. If the cache is large enough, the probability of success, called the hit rate, will be high, and the amount of bus traffic per CPU will drop dramatically, allowing many more CPUs in the system. Cache sizes of 512 KB to 1 MB are common, which often gives a hit rate of 90 percent or more.

![Figure 1-7. A bus-based multiprocessor.](image)

However, the introduction of caches also creates a serious problem. Suppose that two CPUs, A and B, each read the same word into their respective caches. Then A overwrites the word. When B next reads that word, it gets the old value from its cache, not the value A just wrote. The memory is now incoherent, and the system is difficult to program. Caching is also used extensively in distributed systems, and there too we have to deal with the problem of incoherent memory. We return to caching and memory coherence in Chap. 6. For more about bus-based multiprocessors, see Lilja (1993).

The problem with bus-based multiprocessors is their limited scalability, even when using caches. To build a multiprocessor with more than 256 processors, a different method is needed to connect the CPUs with the memory. One possibility is to divide the memory up into modules and connect them to the CPUs with a crossbar switch, as shown in Fig. 1-8(a). Each CPU and each memory has a connection coming out of it, as shown. At every intersection is a tiny electronic crosspoint switch that can be opened and closed in hardware. When a CPU wants to access a particular memory, the crosspoint switch connecting them is closed momentarily, to allow the access to take place. The virtue of the crossbar switch is that many CPUs can be accessing memory at the same time, although if two CPUs try to access the same memory simultaneously, one of them will have to wait.

The downside of the crossbar switch is that with \( n \) CPUs and \( n \) memories, \( n^2 \) crosspoint switches are needed. For large \( n \), this number can be prohibitive. As a result, people have looked for, and found, alternative switching networks that require fewer switches. The omega network of Fig. 1-8(b) is one example. This network contains four \( 2 \times 2 \) switches, each having two inputs and two outputs. Each switch can route either input to either output. A careful look at the figure will show that with proper settings of the switches, every CPU can access every memory. The drawback of switching networks such as these is that there may be...
A cache is like a mini memory. Suppose every processor has its own private cache. Each processor has its own old value cache and its own new cache, and the processor is connected to memory. We have many such memory bus-based systems.

One possibility, even with multiprocessors, is to have a crossbar switch. This has a convenience that no electronic switch is closed when the CPU wants to communicate. If the switch is closed, then one of the CPUs or the switch is faulty. Each CPU can communicate to two CPUs simultaneously without waiting.

There are several switching stages between the CPU and memory. Consequently, to ensure low latency between CPU and memory, switching has to be extremely fast, which is not going to be cheap.

People have attempted to reduce the cost of switching by going to hierarchical systems. Some memory is associated with each CPU. Each CPU can access its own local memory quickly, but accessing anybody else’s memory is slower. This design gives rise to what is known as a NUMA (NonUniform Memory Access) machine. Although NUMA machines have better average access times than machines based on omega networks, they have the new complication that the placement of the programs and data becomes critical in order to make most access go to the local memory.

1.3.2 Homogeneous Multicomputer Systems

In contrast to multiprocessors, building a multicomputer is relatively easy. Each CPU has a direct connection to its own local memory. The only problem left is how the CPUs communicate with each other. Clearly, some interconnection scheme is needed here, too, but since it is only for CPU-to-CPU communication, the volume of traffic will be several orders of magnitude lower than when the interconnection network is also used for CPU-to-memory traffic.

We will first take a look at homogeneous multicomputers. In these systems, which are also referred to as System Area Networks (SANs), the nodes are mounted in a big rack and are connected through a single, often high-performance interconnection network. As before, we make a distinction between systems that are based on a bus, and those that are based on a switch.
In a bus-based multicomputer, the processors are connected through a shared multiaccess network such as Fast Ethernet. The bandwidth of the network is typically 100 Mbps. As with bus-based multiprocessors, bus-based multicomputers have limited scalability. Depending on how much the nodes actually need to communicate, one can generally not expect much performance from systems with more than 25-100 nodes.

In a switch-based multicomputer, messages between the processors are routed through an interconnection network instead of broadcast as in bus-based systems. Many different topologies have been proposed and built. Two popular topologies are meshes and hypercubes, as shown in Fig. 1-9. Grids are easy to understand and lay out on printed circuit boards. They are best suited to problems that have an inherent two-dimensional nature, such as graph theory or vision (e.g., robot eyes or analyzing photographs).

![Figure 1-9. (a) Grid. (b) Hypercube.](image)

A hypercube is an \( n \)-dimensional cube. The hypercube of Fig. 1-9(b) is four-dimensional. It can be thought of as two ordinary cubes, each with 8 vertices and 12 edges. Each vertex is a CPU. Each edge is a connection between two CPUs. The corresponding vertices in each of the two cubes are connected. To expand the hypercube to five dimensions, we would add another set of two interconnected cubes to the figure, connect the corresponding edges in the two halves, and so on.

Switched multicomputers can vary widely. At one end of the spectrum, there are Massively Parallel Processors (MPPs) which are huge, multimillion dollar supercomputers consisting of thousands of CPUs. In many cases, the CPUs are no different than the ones used in workstations or PCs. What makes a difference with other multicomputers is the use of a high-performance proprietary interconnection network. The network is designed to achieve low latency and high bandwidth. Also, special measurements are taken to ensure fault tolerance. With thousands of CPUs, it is inevitable that at least some CPUs will break down every week. It would be unacceptable if a single failing CPU would bring down the entire machine.

At the other end of the spectrum, we find a popular form of switched multicomputers known as Clusters of Workstations (COWs), which are basically a...
collection of standard PCs or workstations connected through off-the-shelf communication components such as Myrinet boards (Boden et al., 1995). It is the interconnection network that distinguishes COWs from MPPs. Also, no special measures are generally taken to ensure high I/O bandwidth or to guard against system failures. By and large, this approach makes COWs simple and cheap.

### 1.3.3 Heterogeneous Multicomputer Systems

Most distributed systems as they are used today are built on top of a heterogeneous multicomputer. This means that the computers that form part of the system may vary widely with respect to, for example, processor type, memory sizes, and I/O bandwidth. In fact, some of the computers may actually be high-performance parallel systems, such as multiprocessors or homogeneous multicomputers.

Also, the interconnection network may be highly heterogeneous as well. As an example, the authors have helped build a home-brew distributed computer system, called DAS, consisting of four clusters of multiprocessors, interconnected through a wide-area ATM-switched backbone. Photos of the system and references to research using it are given at [http://www.cs.vu.nl/~bal/das.html](http://www.cs.vu.nl/~bal/das.html). The clusters can also communicate using standard Internet facilities. Each cluster contains the same CPUs (Pentium III), as well as interconnection network (Myrinet) but varies in their number of processors (64-128).

Another example of heterogeneity is the construction of large-scale multicomputers using existing networks and backbones. For example, it is not uncommon to have a campus-wide distributed system that is running on top of local-area networks from different departments, connected through a high-speed backbone. In wide-area systems, different sites may, in turn, be connected through public networks as offered by commercial carriers using network services such as SMDS or frame relay.

In contrast to the systems discussed in the previous sections, many large-scale, heterogeneous multicomputers lack a global system view, meaning that an application cannot assume that the same performance or services are available everywhere. For example, in the I-way project (Foster and Kesselman, 1998), several high-performance computing centers were interconnected through the Internet. The overall system model was that applications could reserve and use resources at each site, but it was impossible to hide the differences between sites from applications.

Due to their scale, inherent heterogeneity, and most of all, lack of a global system view, sophisticated software is needed to build applications for heterogeneous multicomputers. Here is where distributed systems fit in. To make it unnecessary for application developers to worry about the underlying hardware, distributed systems provide a software layer that shields applications from what is going on at the hardware level (i.e., they provide transparency).
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 for homogeneous
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
execution. In contrast, in **user mode**, memory and register access is restricted. For example, an application will not be allowed to access memory locations that lie outside a range of addresses (set by the operating system), or directly access device registers. While executing operating system code, the CPU is switched to kernel mode. However, the only way to switch from user mode to kernel mode is through system calls as implemented by the operating system. Because system calls are the only basic services an operating system offers, and because the hardware helps to restrict memory and register access, an operating system can be put into full control.

Having two modes of operation has led to organizations of operating systems in which virtually all operating system code is executed in kernel mode. The result is often a huge, monolithic program that is run in a single address space. The drawback of this approach is that it is often difficult to adapt the system. In other words, it is hard to replace or adapt operating system components without doing a complete shutdown and possibly even a full recompilation and reinstallation. Monolithic operating systems are not a good idea from the perspective of openness, software engineering, reliability, or maintainability.

A more flexible approach is to organize the operating system into two parts. The first part consists of a collection of modules for managing the hardware but which can equally well be executed in user mode. For example, memory management basically consists of keeping track of which parts of memory have been allocated to processes, and which parts are free. The only time we need to execute in kernel mode is when the registers of the MMU are set.

The second part of the operating system consists of a small **microkernel** containing only the code that must execute in kernel mode. In practice, a microkernel need only contain the code for setting device registers, switching the CPU between processes, manipulating the MMU, and capturing hardware interrupts. In addition, it contains the code to pass system calls to calls on the appropriate user-level operating system modules, and to return their results. This approach leads to the organization shown in Fig. 1-11.

![Figure 1-11. Separating applications from operating system code through a microkernel.](image)

The microkernel serves as a communication interface between applications and the rest of the operating system. It acts like a network device, passing traffic between applications and the kernel. This allows applications to be run independently and in parallel, without interfering with each other. The microkernel also provides a standardized interface for access to system resources, such as memory and device drivers, which makes it easier to develop and maintain applications.
Their are many benefits to using microkernels. An important one is its flexibility: because a large part of the operating system is executed in user mode, it is relatively easy to replace a module without having to recompile or re-install the entire system. Another important issue is that user-level modules can, in principle, be placed on different machines. For example, we can easily place a file management module on a different machine than the one managing a directory service. In other words, the microkernel approach lends itself well to extending a uniprocessor operating system to distributed computers.

Microkernels have two important disadvantages. First, they are different from the way current operating systems work, and trying to change any well-entrenched status quo always meets massive resistance (“If this operating system is good enough for my grandfather, it is good enough for me.”). Second, microkernels have extra communication and thus a slight performance loss. However, given how fast modern CPUs are, a 20% performance loss is hardly fatal.

**Multiprocessor Operating Systems**

An important, but often not entirely obvious extension to uniprocessor operating systems, is support for multiple processors having access to a shared memory. Conceptually, the extension is simple in that all data structures needed by the operating system to manage the hardware, including the multiple CPUs, are placed into shared memory. The main difference is that these data are now accessible by multiple processors, so they have to be protected against concurrent access to guarantee consistency.

However, many operating systems, especially those for PCs and workstations, cannot easily handle multiple CPUs. The main reason is that they have been designed as monolithic programs that can be executed only with a single thread of control. Adapting such operating systems for multiprocessors generally means redesigning and reimplementing the entire kernel. Modern operating systems are designed from the start to be able to handle multiple processors.

Multiprocessor operating systems aim to support high performance through multiple CPUs. An important goal is to make the number of CPUs transparent to the application. Achieving such transparency is relatively easy because the communication between different (parts of) applications uses the same primitives as those in multitasking uniprocessor operating systems. The idea is that all communication is done by manipulating data at shared memory locations, and that we only have to protect that data against simultaneous access. Protection is done through synchronization primitives. Two important (and equivalent) primitives are semaphores and monitors.

A **semaphore** can be thought of as an integer with two operations, down and up. The down operation checks to see if the value of the semaphore is greater than 0. If so, it decrements its value and continues. If the value is 0, the calling process is blocked. The up operation does the opposite. It first checks whether
there are any now-blocked processes that were unable to complete an earlier down operation. If so, it unblocks one of them and then continues. Otherwise, it simply increments the semaphore value. An unblocked process can simply continue by returning from the down operation. An important property of semaphore operations is that they are atomic, meaning that once a down or up operation has started, no other process can access the semaphore until the operation is completed (or until a process blocks).

Programming with semaphores to synchronize processes is known to be error-prone except when used for simply protecting shared data. The main problem is that the use of semaphores can easily lead to unstructured code, similar to that resulting from abundantly using the infamous goto statement. As an alternative, many modern systems that support concurrent programming provide a library for implementing monitors.

Formally, a monitor is a programming-language construct, similar to an object in object-based programming (Hoare, 1974). A monitor can be thought of as a module consisting of variables and procedures. Variables can be accessed only by calling one of the monitor’s procedures. In this sense, a monitor is similar to an object: an object has its own private data, which can be accessed only by means of methods implemented by that object. The difference with objects, is that a monitor will allow only a single process at a time to execute a procedure. In other words, if a process A is executing a procedure contained in a monitor (we say that A has entered the monitor), and a process B also calls one of the monitor’s procedures, B will be blocked until A completes (i.e., until A leaves the monitor).

As an example, consider a simple monitor for protecting an integer variable as shown in Fig. 1-12. The monitor contains a single (private) variable count that can be accessed only by means of three (public) procedures for respectively reading its current value, incrementing it by 1, or decrementing it. The monitor construct guarantees that any process that calls one of these procedures can atomically access the private data contained in the monitor.

```c
monitor Counter {
    private:
        int count = 0;
    public:
        int value() { return count; }
        void incr() { count = count + 1; }
        void decr() { count = count - 1; }
}
```

Figure 1-12. A monitor to protect an integer against concurrent access.

So far, monitors are useful for simply protecting shared data. However, more is needed for conditionally blocking a process. For example, suppose we wish to
block a process calling the operation decr when it finds out that the value of count has dropped to 0. For such purposes, monitors also contain what is known as condition variables, which are special variables with two operations wait and signal. When process $A$ is inside a monitor, and calls wait on a condition variable contained in that monitor, $A$ will block and give up its exclusive access to the monitor. Consequently, a process $B$ that was waiting to enter the monitor can then continue. At a certain point, $B$ may unblock process $A$ by doing a signal on the condition variable that $A$ is waiting on. To avoid having two processes active inside the monitor, we adopt the scheme by which the signaling process must leave the monitor. We can now adapt our previous example. It can be verified that the monitor shown in Fig. 1-13 is actually an implementation of a semaphore as discussed above.

```
monitor Counter {
    private:
        int count = 0;
        int blocked_procs = 0;
        condition unblocked;

    public:
        int value() { return count; }

        void incr() {
            if (blocked_procs == 0)
                count = count + 1;
            else
                signal(unblocked);
        }

        void decr() {
            if (count == 0) {
                blocked_procs = blocked_procs + 1;
                wait(unblocked);
            } else
                blocked_procs = blocked_procs - 1;
        }

        count = count - 1;
}
```

**Figure 1-13.** A monitor to protect an integer against concurrent access, but blocking a process.

The drawback of monitors is that they are programming-language constructs. For example, Java provides a notion of monitors by essentially allowing each object to protect itself against concurrent access through synchronized statements,
and operations wait and notify on objects. Library support for monitors is generally given by means of simple semaphores that can only take on the values 0 and 1, commonly referred as **mutex variables**, with associated lock and unlock operations. Locking a mutex will succeed only if the mutex is 1, otherwise the calling process will be blocked. Likewise, unlocking a mutex means setting its value to 1, unless some waiting process could be unblocked. Condition variables with their associated operations are also provided as library routines. More information on synchronization primitives can be found in (Andrews, 2000).

**Multicomputer Operating Systems**

Operating systems for multicomputers are of a totally different structure and complexity than multiprocessor operating systems. This difference is caused by the fact that data structures for systemwide resource management can no longer be easily shared by merely placing them in physically shared memory. Instead, the only means of communication is through **message passing**. Multicomputer operating systems are therefore generally organized as shown in Fig. 1-14.

![Multicomputer Operating System Diagram](image)

**Figure 1-14.** General structure of a multicomputer operating system.

Each node has its own kernel containing modules for managing local resources such as memory, the local CPU, a local disk, and so on. Also, each node has a separate module for handling interprocessor communication, that is, sending and receiving messages to and from other nodes.

Above each local kernel is a common layer of software that implements the operating system as a virtual machine supporting parallel and concurrent execution of various tasks. In fact, as we shall discuss shortly, this layer may even provide an abstraction of a multiprocessor machine. In other words, it provides a complete **software implementation** of shared memory. Additional facilities commonly implemented in this layer are, for example, those for assigning a task to a processor, masking hardware failures, providing transparent storage, and general

...
interprocess communication. In other words, facilities that one would normally expect from any operating system.

Multicomputer operating systems that do not provide a notion of shared memory can offer only message-passing facilities to applications. Unfortunately, the semantics of message-passing primitives may vary widely between different systems. It is easiest to explain their differences by considering whether or not messages are buffered. In addition, we need to take into account when, if ever, a sending or receiving process is blocked. Fig. 1-15 shows where buffering and blocking can take place.

![Diagram](image)

**Figure 1-15.** Alternatives for blocking and buffering in message passing.

There are only two places where messages can possibly be buffered: at the sender's side or at the receiver's side. This leads to four possible synchronization points, that is, points at which a sender or receiver can possibly block. If there is a buffer at the sender's side, it makes sense to block the sender only when the buffer is full, which is shown as synchronization point $S1$ in Fig. 1-15. Alternatively, putting a message into a buffer may return a status indicating whether the operation succeeded. This avoids the sender being blocked when the buffer was already full. Otherwise, when there is no sender buffer, there are three alternative points to block the sender: the message has been sent (shown as $S2$), the message has arrived at the receiver (synchronization point $S3$), or the message has been delivered to the receiver (at point $S4$). Note that if blocking takes place at either $S2$, $S3$, or $S4$, having a buffer at the sender's side does not make any sense.

Blocking the receiver makes sense only at synchronization point $S3$, which can happen only when there is no receiver buffer, or when the buffer is empty. An alternative is to let the receiver poll for incoming messages. However, doing so often results in a waste of CPU time, or responding too late to incoming messages, which in turn may lead to buffer overflows resulting in incoming messages having to be dropped (Bhooedjang et al., 1998).

Another issue that is important for understanding message-passing semantics, is whether or not communication is reliable. The distinguishing feature of reliable communication is that the sender is given a guarantee that its messages will be
received. In Fig. 1-15, this means that all messages are guaranteed to make it to synchronization point S3. With unreliable communication, no such guarantee is given. When there is a buffer at the sender's side communication can either be reliable or not. Likewise, the operating system need not guarantee reliable communication when the sender is blocked at S2.

However, if the operating system blocks a sender until messages arrive at either S3 or S4, it must guarantee reliable communication, or we may otherwise find ourselves in a situation in which the sender is waiting for confirmation of receipt or delivery, while in the meantime its message had been lost during transmission. The relations between blocking, buffering, and guarantees regarding reliable communication are summarized in Fig. 1-16.

<table>
<thead>
<tr>
<th>Synchronization point</th>
<th>Send buffer</th>
<th>Reliable comm. guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block sender until buffer not full</td>
<td>Yes</td>
<td>Not necessary</td>
</tr>
<tr>
<td>Block sender until message sent</td>
<td>No</td>
<td>Not necessary</td>
</tr>
<tr>
<td>Block sender until message received</td>
<td>No</td>
<td>Necessary</td>
</tr>
<tr>
<td>Block sender until message delivered</td>
<td>No</td>
<td>Necessary</td>
</tr>
</tbody>
</table>

*Figure 1-16. Relation between blocking, buffering, and reliable communication.*

Many of the issues involved in building multicomputer operating systems are equally important for any distributed system. The main difference between multicomputer operating systems and distributed systems is that the former generally assume that the underlying hardware is homogeneous and is to be fully managed. Many distributed systems, however, are often built on top of existing operating systems, as we will discuss shortly.

**Distributed Shared Memory Systems**

Practice shows that programming multicomputers is much harder than programming multiprocessors. The difference is caused by the fact that expressing communication in terms of processes accessing shared data and using simple synchronization primitives like semaphores and monitors is much easier than having only message-passing facilities available. Issues like buffering, blocking, and reliable communication only make things worse.

For this reason, there has been considerable research in emulating shared-memory on multicomputers. The goal is to provide a virtual shared memory machine, running on a multicomputer, for which applications can be written using the shared memory model even though this is not present. The multicomputer operating system plays a crucial role here.

One approach is to use the virtual memory capabilities of each individual node to support a large virtual address space. This leads to what is called a page-
based distributed shared memory (DSM). The principle of page-based distributed shared memory is as follows. In a DSM system, the address space is divided up into pages (typically 4 KB or 8 KB), with the pages being spread over all the processors in the system. When a processor references an address that is not present locally, a trap occurs, and the operating system fetches the page containing the address and restarts the faulting instruction, which now completes successfully. This concept is illustrated in Fig. 1-17(a) for an address space with 16 pages and four processors. It is essentially normal paging, except that remote RAM is being used as the backing store instead of the local disk.

![Diagram of shared global address space]

**Figure 1-17.** (a) Pages of address space distributed among four machines. (b) Situation after CPU 1 references page 10. (c) Situation if page 10 is read only and replication is used.

In this example, if processor 1 references instructions or data in pages 0, 2, 5, or 9, the references are done locally. References to other pages cause traps. For
example, a reference to an address in page 10 will cause a trap to the operating system, which then moves page 10 from machine 2 to machine 1, as shown in Fig. 1-17(b).

One improvement to the basic system that can frequently improve performance considerably is to replicate pages that are read only, for example, pages that contain program text, read-only constants, or other read-only data structures. For example, if page 10 in Fig. 1-17 is a section of program text, its use by processor 1 can result in a copy being sent to processor 1, without the original in processor 2's memory being disturbed, as shown in Fig. 1-17(c). In this way, processors 1 and 2 can both reference page 10 as often as needed without causing traps to fetch missing memory.

Another possibility is to replicate not only read-only pages, but all pages. As long as reads are being done, there is effectively no difference between replicating a read-only page and replicating a read-write page. However, if a replicated page is suddenly modified, special action has to be taken to prevent having multiple, inconsistent copies in existence. Typically all copies but one are invalidated before allowing the write to proceed.

Further performance improvements can be made if we let go of strict consistency between replicated pages. In other words, we allow a copy to be temporarily different from the others. Practice has shown that this approach may indeed help, but unfortunately, can also make life much harder for the programmer as he has to be aware of such inconsistencies. Considering that ease of programming was an important reason for developing DSM systems in the first place, weakening consistency may not be a real alternative. We return to consistency issues in Chap. 6.

Another issue in designing efficient DSM systems, is deciding how large pages should be. Here, we are faced with similar trade-offs as in deciding on the size of pages in uniprocessor virtual memory systems. For example, the cost of transferring a page across a network is primarily determined by the cost of setting up the transfer and not by the amount of data that is transferred. Consequently, having large pages may possibly reduce the total number of transfers when large portions of contiguous data need to be accessed. On the other hand, if a page contains data of two independent processes on different processors, the operating system may need to repeatedly transfer the page between those two processors, as shown in Fig. 1-18. Having data belonging to two independent processes in the same page is called \textbf{false sharing}.

After almost 15 years of research on distributed shared memory, DSM researchers are still struggling to combine efficiency and programmability. To attain high performance on large-scale multicomputers, programmers resort to message passing despite its higher complexity compared to programming (virtual) shared memory systems. It seems therefore justified to conclude that DSM for high-performance parallel programming cannot fulfill its initial expectations. More information on DSM can be found in (Protic et al., 1998).
1.4.2 Network Operating Systems

In contrast to distributed operating systems, network operating systems do not assume that the underlying hardware is homogeneous and that it should be managed as if it were a single system. Instead, they are generally constructed from a collection of uniprocessor systems, each with its own operating system, as shown in Fig. 1-19. The machines and their operating systems may be different, but they are all connected to each other in a computer network. Also, network operating systems provide facilities to allow users to make use of the services available on a specific machine. It is perhaps easiest to describe network operating systems by taking a closer look at some services they typically offer.

![Figure 1-19. General structure of a network operating system.](image)

A service that is commonly provided by network operating systems is to allow a user to log into another machine remotely by using a command such as

```
login machine
```

The effect of this command is to turn the user’s own workstation into a remote terminal logged into the remote machine. Assuming the user is sitting behind a
INTRODUCTION

Graphical workstation, commands typed on the keyboard are sent to the remote machine, and output from the remote machine is displayed in a window on the user’s screen. To switch to a different remote machine, it is necessary to first open a new window, and then to use the rlogin command to connect to another machine. The selection of the machine is thus entirely manual.

Network operating systems often also have a remote copy command to copy files from one machine to another. For example, a command such as

```
rcp machine1:file1 machine2:file2
```

might copy the file `file1` from `machine1` to `machine2` and give the name `file2` there. Again here, the movement of files is explicit and requires the user to be completely aware of where all files are located and where all commands are being executed.

While better than nothing, this form of communication is extremely primitive and has led system designers to search for more convenient forms of communication and information sharing. One approach is to provide a shared, global file system accessible from all the workstations. The file system is supported by one or more machines called file servers. The file servers accept requests from user programs running on the other (nonserver) machines, called clients, to read and write files. Each incoming request is examined and executed, and the reply is sent back, as illustrated in Fig. 1-20.

![Figure 1-20. Two clients and a server in a network operating system.](image)

File servers generally maintain hierarchical file systems, each with a root directory containing subdirectories and files. Workstations can import or mount these file systems, augmenting their local file systems with those located on the servers. For example, in Fig. 1-21, two file servers are shown. One has a directory called `games`, while the other has a directory called `work` (directory names are shown in boldface). These directories each contain several files. Both of the clients shown have mounted both of the servers, but they have mounted them in different places in their respective file systems. Client 1 has mounted them in its root directory, and can access them as `games` and `work`, respectively. Client 2, like client 1, has mounted `work` in its root directory, but considers playing games
as something that should perhaps be kept private. It therefore created a directory called `/private` and mounted `games` there. Consequently, it can access `pacwoman` using the path `/private/games/pacwoman` rather than `/games/pacwoman`.

![Diagram](image)

**Figure 1-21.** Different clients may mount the servers in different places.

While it often does not matter where a client mounts a server in its directory hierarchy, it is important to notice that different clients can have a different view of the file system. The name of a file depends on where it is being accessed from, and how that machine has set up its file system. Because each client machine operates relatively independently of the others, there is no guarantee that they all present the same directory hierarchy to their programs.

Network operating systems are clearly more primitive than distributed operating systems. The main distinction between the two types of operating systems is that distributed operating systems make a serious attempt to realize full transparency, that is, provide a single-system view.

The lack of transparency in network operating systems has some obvious drawbacks. For example, they are often harder to use, as users are required to explicitly log into remote machines, or copy files from one machine to another. There is also a management problem. Because all machines in a network operating system are independent, often they can only be managed independently. As a consequence, a user can do a remote login to a machine X only if he has an account on X. Likewise, if a user wants to use only a single password, changing a password requires changing it explicitly on every machine. In the same line of reasoning, it is seen that, in general, all access permissions have to be maintained per machine as well. There is no simple way of changing permissions once they
are the same everywhere. This decentralized approach to security sometimes makes it hard to protect network operating systems against malicious attacks.

There are also some advantages compared to distributed operating systems. As the nodes in a network operating system are highly independent of each other, it is easy to add or remove a machine. In some cases, the only thing we need to do to add a machine is to connect the machine to a common network and, subsequently, make its existence known to the other machines in that network. In the Internet, for example, adding a new server is done precisely in this way. To make a machine known across the Internet, we need merely provide its network address, or better, give the machine a symbolic name that we subsequently enter into DNS, along with its network address.

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.
Software Concepts

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
rigorously followed in Plan 9 (Pike et al., 1995). In Plan 9, all resources, including I/O devices such as keyboard, mouse, disk, network interface, and so on, are treated as files. Essentially, whether a file is local or remote makes no difference. An application opens a file, reads and writes bytes, and closes it again. Because files can be shared by several processes, communication reduces to simply accessing the same file.

A similar approach, but less strict than in Plan 9, is followed by middleware centered around distributed file systems. In many cases, such middleware is actually only one step beyond a network operating system in the sense that distribution transparency is supported only for traditional files (i.e., files that are used for merely storing data). For example, processes are often required to be started explicitly on specific machines. Middleware based on distributed file systems has proven to be reasonable scalable, which contributes to its popularity.

Another important early middleware model is that based on Remote Procedure Calls (RPCs). In this model, the emphasis is on hiding network communication by allowing a process to call a procedure of which an implementation is located on a remote machine. When calling such a procedure, parameters are transparently shipped to the remote machine where the procedure is subsequently executed, after which the results are sent back to the caller. It therefore appears as if the procedure call was executed locally: the calling process remains unaware of the fact that network communication took place, except perhaps for some loss of performance. We return to remote procedure calls in the next chapter.

As object orientation came into vogue, it became apparent that if procedure calls could cross machine boundaries, it should also be possible to invoke objects residing on remote machines in a transparent fashion. This has now led to various middleware systems offering a notion of distributed objects. The essence of distributed objects is that each object implements an interface that hides all the internal details of the object from its users. An interface consists of the methods that the object implements, no more and no less. The only thing that a process sees of an object is its interface.

Distributed objects are often implemented by having each object itself located on a single machine, and additionally making its interface available on many other machines. When a process invokes a method, the interface implementation on the process’s machine simply transforms the method invocation into a message that is sent to the object. The object executes the requested method and sends back the result. The interface implementation subsequently transforms the reply message into a return value, which is then handed over to the invoking process. As in the case of RPC, the process may be kept completely unaware of the network communication.

What models can do to simplify the use of networked systems is probably best illustrated by the World Wide Web. The success of the Web is mainly due to the extremely simple, yet highly effective model of distributed documents. In the model of the Web, information is organized into documents, with each document
residing at a machine transparently located somewhere in the world. Documents contain links that refer to other documents. By following a link, the document to which that link refers is fetched from its location and displayed on the user's screen. The concept of a document need not be restricted to only text-based information. For example, the Web also supports audio and video documents, as well as all kinds of interactive graphic-based documents.

We return to middleware paradigms extensively in the second part of the book.

### Middleware Services

There are a number of services common to many middleware systems. Invariably, all middleware, one way or another, attempts to implement access transparency, by offering high-level communication facilities that hide the low-level message passing through computer networks. The programming interface to the transport layer as offered by network operating systems is thus entirely replaced by other facilities. How communication is supported depends very much on the model of distribution the middleware offers to users and applications. We already mentioned remote procedure calls and distributed-object invocations. In addition, many middleware systems provide facilities for transparent access to remote data, such as distributed file systems or distributed databases. Transparently fetching documents as is done in the Web is another example of high-level (one-way) communication.

An important service common to all middleware is that of **naming**. Name services allow entities to be shared and looked up (as in directories), and are comparable to telephone books and the yellow pages. Although naming may seem simple at first thought, difficulties arise when scalability is taken into account. Problems are caused by the fact that to efficiently look up a name in a large-scale system, the location of the entity that is named must be assumed to be fixed. This assumption is made in the World Wide Web, in which each document is currently named by means of a URL. A URL contains the name of the server where the document to which the URL refers is stored. Therefore, if the document is moved to another server, its URL ceases to work.

Many middleware systems offer special facilities for storage, also referred to as **persistence**. In its simplest form, persistence is offered through a distributed file system, but more advanced middleware have integrated databases into their systems, or otherwise provide facilities for applications to connect to databases.

In environments where data storage plays an important role, facilities are generally offered for **distributed transactions**. An important property of a transaction is that it allows multiple read and write operations to occur atomically. Atomicity means that the transaction either succeeds, so that all its write operations are actually performed, or it fails, leaving all referenced data unaffected. Distributed transactions operate on data that are possibly spread across multiple machines.
Especially in the face of masking failures, which is often hard in distributed systems, it is important to offer services such as distributed transactions. Unfortunately, transactions are hard to scale across many local machines, let alone geographically dispersed machines.

Finally, virtually all middleware systems that are used in nonexperimental environments provide facilities for security. Compared to network operating systems, the problem with security in middleware is that it should be pervasive. In principle, the middleware layer cannot rely on the underlying local operating systems to adequately support security for the complete network. Consequently, security has to be partly implemented anew in the middleware layer itself. Combined with the need for extensibility, security has turned out to be one of the hardest services to implement in distributed systems.

**Middleware and Openness**

Modern distributed systems are generally constructed as middleware for a range of operating systems. In this way, applications built for a specific distributed system become operating system independent. Unfortunately, this independence is often replaced by a strong dependency on specific middleware. Problems are caused by the fact that middleware is often less open than claimed.

As we explained previously, a truly open distributed system is specified by means of interfaces that are complete. Complete means that everything that is needed for implementing the system, has indeed been specified. Incompleteness of interface definitions leads to the situation in which system developers may be forced to add their own interfaces. Consequently, we may end up in a situation in which two middleware systems from different development teams adhere to the same standard, but applications written for one system cannot be easily ported to the other.

Equally bad is the situation in which incompleteness leads to a situation in which two different implementations can never interoperate, despite the fact that they implement exactly the same set of interfaces but different underlying protocols. For example, if two different implementations rely on incompatible communication protocols as available in the underlying network operating system, there is little hope that interoperability can be easily achieved. What we need is that middleware protocols and the interfaces to the middleware are the same, as shown in Fig. 1-23.

As another example, to ensure interoperability between different implementations, it is necessary that entities within the different systems are referenced in the same way. If entities in one system are referenced by means of URLs, while the other system implements references using network addresses, it is clear that cross referencing is going to be a problem. In such cases, the interface definitions should have prescribed precisely what references look like.
A brief comparison between distributed operating systems, network operating systems, and (middleware-based) distributed systems is given in Fig. 1-24.

<table>
<thead>
<tr>
<th>Item</th>
<th>Distributed OS</th>
<th>Network OS</th>
<th>Middleware-based DS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multiproc.</td>
<td>Multicomp.</td>
<td>Low</td>
</tr>
<tr>
<td>Degree of transparency</td>
<td>Very high</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Same OS on all nodes?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Number of copies of OS</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Basis for communication</td>
<td>Shared memory</td>
<td>Messages</td>
<td>Files</td>
</tr>
<tr>
<td>Resource management</td>
<td>Global, central</td>
<td>Global, distributed</td>
<td>Per node</td>
</tr>
<tr>
<td>Scalability</td>
<td>No</td>
<td>Moderately</td>
<td>Yes</td>
</tr>
<tr>
<td>Openness</td>
<td>Closed</td>
<td>Closed</td>
<td>Open</td>
</tr>
</tbody>
</table>

Figure 1-23. In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.

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-computer operating systems that support full distribution transparency is so difficult. Distributed systems often improve transparency by adopting a specific model for distribution and communication. For example, distributed file systems are generally good at hiding the location and access to files. However, they lose some generality as users are forced to express everything in terms of that specific model, which may be sometimes inconvenient for a specific application.
Distributed operating systems are homogeneous, implying that each node runs the same operating system (kernel). In multiprocessor systems, no copies of tables and such are needed, as they can all be shared through main memory. In this case, all communication also happens through main memory, whereas in multicomputer operating systems messages are used. In network operating systems, one could argue that communication is almost entirely file based. For example, in the Internet, a lot of communication is done by transferring files. However, high-level messaging in the form of electronic mail systems and bulletin boards is also used extensively. Communication in middleware-based distributed systems depends on the model specifically adopted by the system.

Resources in network operating systems and distributed systems are managed per node, which makes such systems relatively easy to scale. However, practice shows that an implementation of the middleware layer in distributed systems often has limited scalability. Distributed operating systems have global resource management, making them harder to scale. Because of the centralized approach in multiprocessor systems (i.e., all management data is kept in main memory), these systems are often hard to scale.

Finally, network operating systems and distributed systems win when it 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.

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.](image)

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.

CHAP. 1

SEC. 1.5

THE CLIENT-SERVER MODEL

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.](image)

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.

CHAP. 1

SEC. 1.5

THE CLIENT-SERVER MODEL

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.](image)

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.
An Example Client and Server

To provide more insight into how clients and servers work, in this section we present an outline of a client and a file server in C. Both the client and the server need to share some definitions, so we will collect these into a file called header.h, which is shown in Fig. 1-26. Both the client and server include these definitions using the

```c
#include <header.h>
```

statement. This statement has the effect of causing a preprocessor to literally insert the entire contents of header.h into the source program just before the compiler starts compiling the program.

```c
/* Definitions needed by clients and servers. */
#define TRUE 1
#define MAX_PATH 255 /* maximum length of file name */
#define BUF_SIZE 1024 /* how much data to transfer at once */
#define FILE_SERVER 243 /* file server's network address */

/* Definitions of the allowed operations */
#define CREATE 1 /* create a new file */
#define READ 2 /* read data from a file and return it */
#define WRITE 3 /* write data to a file */
#define DELETE 4 /* delete an existing file */

/* Error codes. */
#define OK 0 /* operation performed correctly */
#define BAD_OPER -1 /* unknown operation requested */
#define BAD_PARAM -2 /* error in a parameter */
#define BAD_IO -3 /* disk error or other I/O error */

/* Definition of the message format. */
struct message {
  long source; /* sender's identity */
  long dest; /* receiver's identity */
  long opcode; /* requested operation */
  long count; /* number of bytes to transfer */
  long offset; /* position in file to start I/O */
  long result; /* result of the operation */
  char name[MAX_PATH]; /* name of file being operated on */
  char data[BUF_SIZE]; /* data to be read or written */
};
```

Figure 1-26. The header.h file used by the client and server.
SEC. 1.5 THE CLIENT-SERVER MODEL

Let us first take a look at header.h. It starts out by defining two constants, MAX_PATH and BUF_SIZE, that determine the size of two arrays needed in the message. The former tells how many characters a file name (i.e., a path name like /usr/ast/books/opsys/chapter1.t) may contain. The latter fixes the amount of data that may be read or written in one operation by setting the buffer size. The next constant, FILE_SERVER, provides the network address of the file server so that clients can send messages to it.

The second group of constants defines the operation numbers. These are needed to ensure that the client and server agree on which code will represent a read, which code will represent a write, and so on. We have shown only four here, but in a real system there would normally be more.

Every reply contains a result code. If the operation succeeds, the result code often contains useful information (such as the number of bytes actually read). If there is no value to be returned (such as when a file is created), the value OK is used. If the operation is unsuccessful for some reason, the result code tells why, using codes such as E_BAD_OPER, E_BAD_PARAM, and so on.

Finally, we come to the most important part of header.h, the definition of the message itself. In our example it is a structure with 8 fields. All requests from the client to the server use this format, as do all replies. In a real system, one would probably not have a fixed format message (because not all the fields are needed in all cases), but it makes the explanation simpler here. The source and dest fields identify the sender and receiver, respectively. The opcode field is one of the operations defined above, that is, create, read, write, or delete. The count and offset fields are used for parameters. The result field is not used for client-to-server requests but holds the result value for server-to-client replies. Finally, we have two arrays. The first, name, holds the name of the file being accessed. The second, data, holds the data sent back on a reply to read or the data sent to the server on a write.

Let us now look at the code, as outlined in Fig. 1-27. In (a) we have the server; in (b) we have the client. The server is straightforward. The main loop starts out by calling receive to get a request message. The first parameter identifies the caller by giving its address, and the second parameter points to a message buffer where the incoming message can be stored. The procedure receive blocks the server until a message arrives. When one comes in, the server continues and dispatches on the opcode type. For each opcode, a different procedure is called. The incoming message and a buffer for the outgoing message are given as parameters. The procedure examines the incoming message, \( m1 \), and builds the reply in \( m2 \). It also returns a function value that is sent back in the result field. After the send has completed, the server goes back to the top of the loop to execute receive and wait for the next incoming message.

In Fig. 1-27(b) we have a procedure that copies a file using the server. The body of the procedure consists of a loop that reads one block from the source file and writes it to the destination file. The loop is repeated until the source file has
been copied completely. The latter is indicated by the return code from the read, which can be a zero or negative value.

The first part of the loop is concerned with building a message for the read operation and sending it to the server. After the reply has been received, the second part of the loop is entered, which takes the data just received and sends it back to the server in the form of a write to the destination file. The programs of Fig. 1-27 are just sketches of the code. Many details have been omitted. For example, the do.xxx procedures (the ones that actually do the work) are not shown, and no error checking is done. Still, the general idea of how a client and a server interact should be clear. In the following sections we take a closer look at 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
#include <header.h>
void main(void) {
    struct message ml, m2; /* incoming/outgoing messages */
    int r; /* result code */
    while(TRUE) { /* server runs forever */
        receive(FILE_SERVER, &ml); /* block waiting for a message */
        switch(ml.opcode) { /* dispatch on type of request */
            case CREATE: r = do_create(&ml, &m2); break; /* r = do_read(&ml, &m2); break; */
            case READ: r = do_read(&ml, &m2); break; /* r = do_write(&ml, &m2); break; */
            case WRITE: r = do_write(&ml, &m2); break; /* r = do_delete(&ml, &m2); break; */
            case DELETE: r = do_delete(&ml, &m2); break; /* r = E_BAD_OP ; */
            default:;
        }
        m2.result = r; /* return result to client */
        send(ml.source, &m2); /* send reply */
    }
}

#include <header.h>
int copy(char *src, char *dst){ /* proc. to copy file using the server */
    struct message ml;
    long position;
    long client = 110;
    initialize();
    position = 0;
    do {
      ml.opcode = READ; /* operation is a read */
      ml.offset = position; /* current position in the file */
      long ml.count = BUF_SIZE; /* how many bytes to read */
      strcpy(&ml.name, src); /* copy name of file to be read */
      send(FILESERVER, &ml); /* send message to the file server */
      receive(client, &ml); /* block waiting for the reply */

      /* Write the data just received to the destination file. */
      ml.opcode = WRITE; /* operation is a write */
      ml.offset = position; /* current position in the file */
      long ml.count = ml.result; /* how many bytes to write */
      strcpy(&ml.name, dst); /* copy name of file to be written */
      send(FILE_SERVER, &ml); /* send message to the file server */
      receive(client, &ml); /* block waiting for the reply */
      position += ml.result; /* ml.result is #bytes written */
    } while( ml.result > 0 ); /* iterate until done */
    return(ml.result >= 0 ? OK : ml.result); /* return OK or error code */
}

Figure 1-27. (a) A sample server. (b) A client using that server to copy a file.
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.
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 distribution decision regarding to what extent...
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.

![Diagram of client-server organizations]

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.
INTRODUCTION

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.

![Diagram showing three-tier architecture](image)

**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.

![Diagram of horizontal distribution of a Web service](image)

**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?
PROBLEMS

4. Why is it not always a good idea to aim at implementing the highest degree of transparency possible?

5. What is an open distributed system and what benefits does openness provide?

6. Describe precisely what is meant by a scalable system.

7. Scalability can be achieved by applying different techniques. What are these techniques?

8. What is the difference between a multiprocessor and a multicomputer?

9. A multicomputer with 256 CPUs is organized as a 16 × 16 grid. What is the worst-case delay (in hops) that a message might have to take?

10. Now consider a 256-CPU hypercube. What is the worst-case delay here, again in hops?

11. What is the difference between a distributed operating system and a network operating system?

12. Explain how microkernels can be used to organize an operating system in a client-server fashion.

13. Explain the principal operation of a page-based distributed shared memory system.

14. What is the reason for developing distributed shared memory systems? What do you see as the main problem hindering efficient implementations?

15. Explain what false sharing is in distributed shared memory systems. What possible solutions do you see?

16. An experimental file server is up 3/4 of the time and down 1/4 of the time, due to bugs. How many times does this file server have to be replicated to give an availability of at least 99 percent?

17. What is a three-tiered client-server architecture?

18. What is the difference between a vertical distribution and a horizontal distribution?

19. Consider a chain of processes \( P_1, P_2, \ldots, P_n \) implementing a multitiered client-server architecture. Process \( P_i \) is client of process \( P_{i+1} \), and \( P_i \) will return a reply to \( P_{i-1} \) only after receiving a reply from \( P_{i+1} \). What are the main problems with this organization when taking a look at the request-reply performance at process \( P_1 \)?