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

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

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

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

Uniprocessor Operating Systems

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

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

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

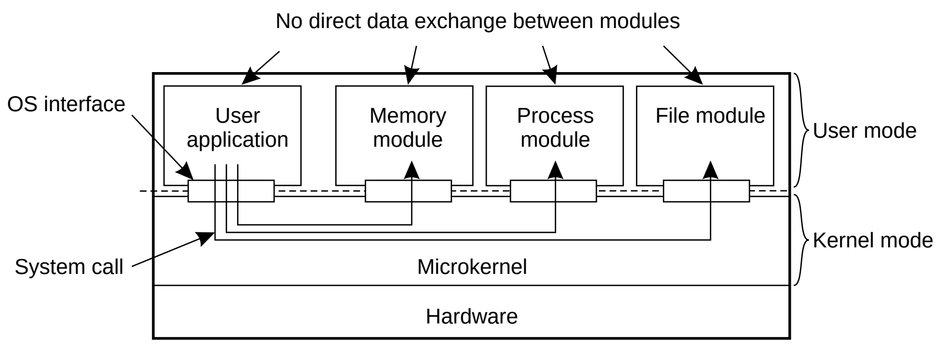

Figure 1-11. Separating applications from operating system code through a microkernel.

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 that 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.

```
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 );
                blocked_procs = blocked_procs – 1;
        }
        else
                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.

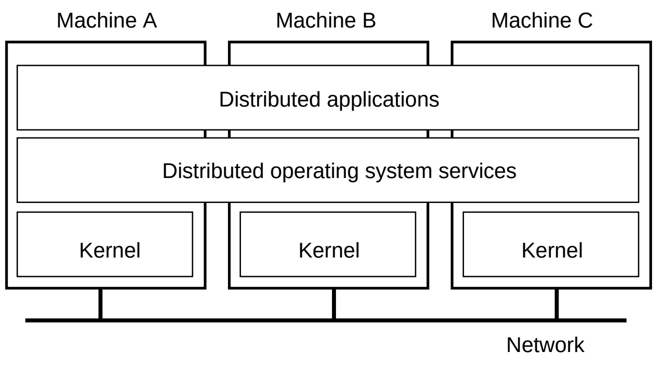

Figure 1-14. General structure of a multicomputer operating system.

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.

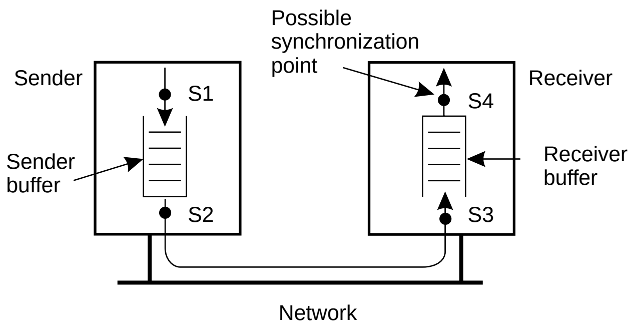

Figure 1-15. Alternatives for blocking and buffering in message passing.

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 (Bhoedjang 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.

Synchronization point	Send buffer	Reliable comm. guaranteed?
Block sender until buffer not full	Yes	Not necessary
Block sender until message sent	No	Not necessary
Block sender until message received	No	Necessary
Block sender until message delivered	No	Necessary

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.

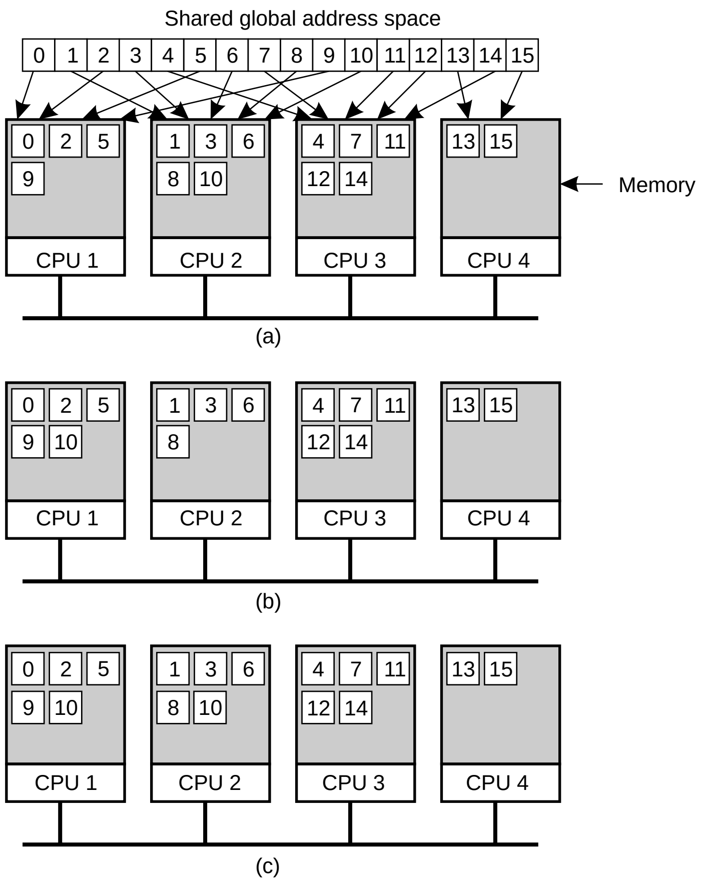

Figure 1-17. (a) Pages of address space distributed among four machines. (b) Situation after CPU 1 references page 10. (c) Situation if page 10 is read only and replication is used.

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 **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).

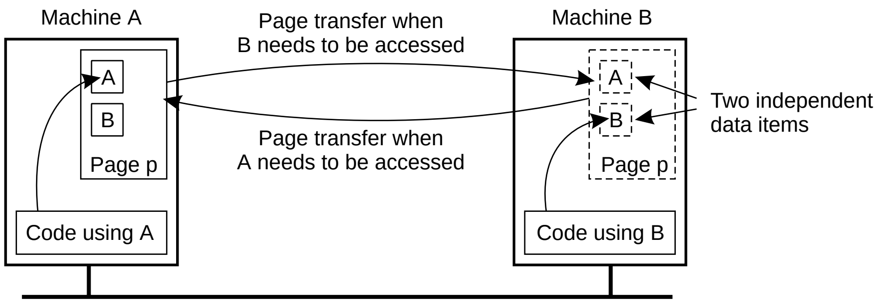
Machine A
Page transfer when B needs to be accessed
Machine B
A
B
Page p
Page transfer when A needs to be accessed
Two independent data items
Code using A
Code using B

Figure 1-18. False sharing of a page between two independent processes.

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.

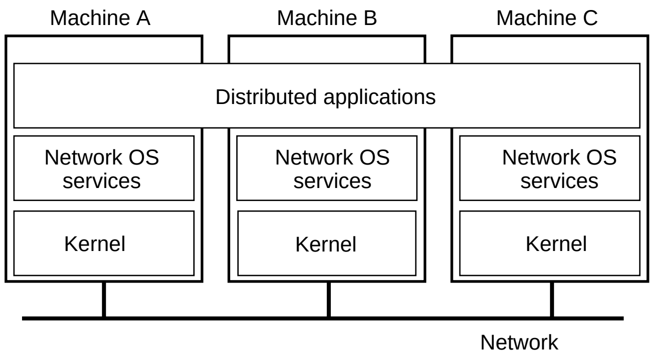

Figure 1-19. General structure of a network operating system.

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

```
rlogin 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 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 it 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.

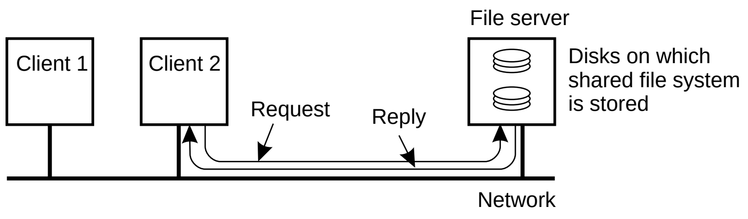

Figure 1-20. Two clients and a server in a network operating system.

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*.

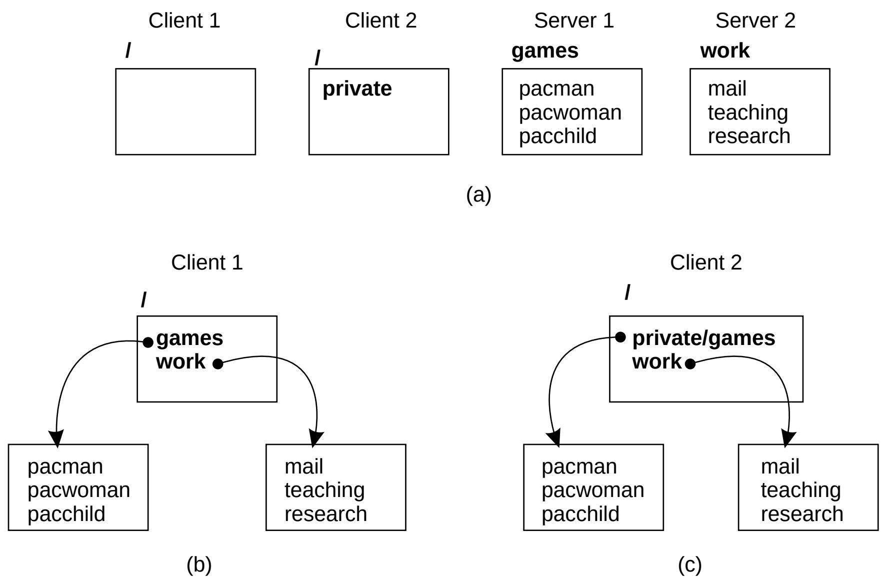

Figure 1-21. Different clients may mount the servers in different places.

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.

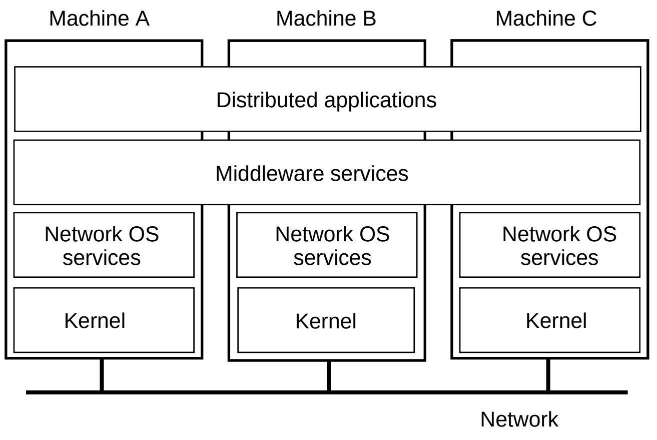

Figure 1-22. General structure of a distributed system as middleware.

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

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

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

Middleware Models

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

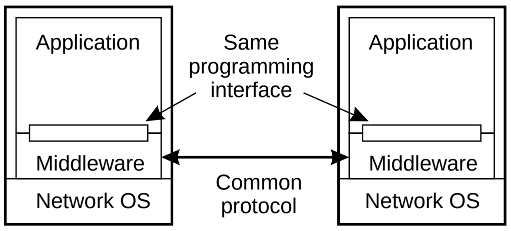

Figure 1-23. In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.

A Comparison between Systems

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

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

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

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