3

PROCESSES

In this chapter, we take a closer look at how the different types of processes playa crucial role in distributed systems. The concept of a process originates from the field of operating systems where it is generally defined as a program in execution. From an operating-system perspective, the management and scheduling of processes are perhaps the most important issues to deal with. However, when it comes to distributed systems, other issues tum out to be equally or more important.

For example, to efficiently organize client-server systems, it is often convenient to make use of multithreading techniques. As we discuss in the first section, a main contribution of threads in distributed systems is that they allow clients and servers to be constructed such that communication and local processing can overlap, resulting in a high level of performance.

In recent years, the concept of virtualization has gained popularity. Virtualization allows an application, and possibly also its complete environment including the operating system, to run concurrently with other applications, but highly independent of the underlying hardware and platforms, leading to a high degree of portability. Moreover, virtualization helps in isolating failures caused by errors or security problems. It is an important concept for distributed systems, and we pay attention to it in a separate section.

As we argued in Chap. 2, client-server organizations are important in distributed systems. In this chapter, we take a closer look at typical organizations of both clients and servers. We also pay attention to general design issues for servers.

An important issue, especially in wide-area distributed systems, is moving processes between different machines. Process migration or more specifically, code migration, can help in achieving scalability, but can also help to dynamically configure clients and servers. What is actually meant by code migration and what its implications are is also discussed in this chapter.

3.1 THREADS

Although processes form a building block in distributed systems, practice indicates that the granularity of processes as provided by the operating systems on which distributed systems are built is not sufficient. Instead, it turns out that having a finer granularity in the form of multiple threads of control per process makes it much easier to build distributed applications and to attain better performance. In this section, we take a closer look at the role of threads in distributed systems and explain why they are so important. More on threads and how they can be used to build applications can be found in Lewis and Berg (998) and Stevens (1999).

3.1.1 Introduction to Threads

To understand the role of threads in distributed systems, it is important to understand what a process is, and how processes and threads relate. To execute a program, an operating system creates a number of virtual processors, each one for running a different program. To keep track of these virtual processors, the operating system has a process **table,** containing entries to store CPU register values, memory maps, open files, accounting information. privileges, etc. A process is often defined as a program in execution, that is, a program that is currently being executed on one of the operating system's virtual processors. An important issue is that the operating system takes great care to ensure that independent processes cannot maliciously or inadvertently affect the correctness of each other's behavior. In other words, the fact that multiple processes may be concurrently sharing the same CPU and other hardware resources is made transparent. Usually, the operating system requires hardware support to enforce this separation.

This concurrency transparency comes at a relatively high price. For example, each time a process is created, the operating system must create a complete independent address space. Allocation can mean initializing memory segments by, for example, zeroing a data segment, copying the associated program into a text segment, and setting up a stack for temporary data. Likewise, switching the CPU between two processes may be relatively expensive as well. Apart from saving the CPU context (which consists of register values, program counter, stack pointer, etc.), the operating system will also have to modify registers of the memory management unit (MMU) and invalidate address translation caches such as in the translation lookaside buffer (TLB). In addition, if the operating system supports

more processes than it can simultaneously hold in main memory, it may have to swap processes between main memory and disk before the actual switch can take place.

Like a process, a thread executes its own piece of code, independently from other threads. However, in contrast to processes, no attempt is made to achieve a high degree of concurrency transparency if this would result in performance degradation. Therefore, a thread system generally maintains only the minimum information to allow a CPU to be shared by several threads. In particular, a thread context often consists of nothing more than the CPU context, along with some other information for thread management. For example, a thread system may keep track of the fact that a thread is currently blocked on a mutex variable, so as not to select it for execution. Information that is not strictly necessary to manage multiple threads is generally ignored. For this reason, protecting data against inappropriate access by threads within a single process is left entirely to application developers.

There are two important implications of this approach. First of all, the performance of a multithreaded application need hardly ever be worse than that of its single-threaded counterpart. In fact, in many cases, multithreading leads to a performance gain. Second, because threads are not automatically protected against each other the way processes are, development of multithreaded applications requires additional intellectual effort. Proper design and keeping things simple, as usual, help a lot. Unfortunately, current practice does not demonstrate that this principle is equally well understood.

Thread Usage in Nondistributed Systems

Before discussing the role of threads in distributed systems, let us first consider their usage in traditional, nondistributed systems. There are several benefits to multithreaded processes that have increased the popularity of using thread systems.

i'ne most 1:m-poron\\)e'i\'tl\\ \.~'m.'t~'i.\)\\\.~ \~\ \\\.~\ \.~ ~ ~\.~¥,t~-t.N~d ~t()c-ess., ~1l.~~~'l:~~a. l:1lQ.c.kiu.~&'!&temcall is executed. tile Qrocess as a wriore *is* MocKea'. 10 Illustrate, corrsrirer Jff <1flfllic«ti<Jt7 s~k cZScZ s~e.2dshc>e!prOgE.wlJ, , a,mj *asscattc tkat« «sercootioUOlIS)Y.:md lZ;!cEacJ)ve)y w..avts JD!''h.ange* values, An important property of a spreadsheet program is that It maintains the runcnonai dependencies between different cells, often from different spreadsheets. Therefore, whenever a cell is modified, all dependent cells are automatically updated. When a user changes the value in a single cell, such a modification can trigger a large series of computations. If there is only a single thread of control, computation cannot proceed while the program is waiting for input. Likewise, it is not easy to provide input while dependencies are being calculated. The easy solution is to have at least two threads of control: one for handling interaction with the user and

one for updating the spreadsheet. In the mean time, a third thread could be used for backing up the spreadsheet to disk while the other two are doing their work.

Another advantage of multithreading is that it becomes possible to exploit parallelism when executing the program on a multiprocessor system. In that case, each thread is assigned to a different CPU while shared data are stored in shared main memory. When properly designed, such parallelism can be transparent: the process will run equally well on a uniprocessor system, albeit slower. Multithreading for parallelism is becoming increasingly important with the availability of relatively cheap multiprocessor workstations. Such computer systems are typically used for running servers in client-server applications.

Multithreading is also useful in the context of large applications. Such applications are often developed as a collection of cooperating programs, each to be executed by a separate process. This approach is typical for a UNIX environment. Cooperation between programs is implemented by means of interprocess communication (IPC) mechanisms. For UNIX systems, these mechanisms typically include (named) pipes, message queues, and shared memory segments [see also Stevens and Rago (2005)]. The major drawback of all IPC mechanisms is that communication often requires extensive context switching, shown at three different points in Fig. 3-1.

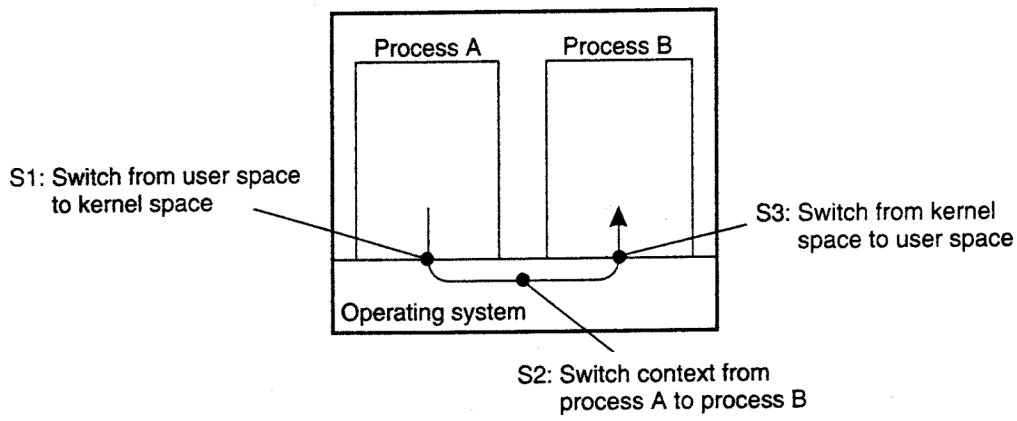

Figure 3-1. Context switching as the result of IPC.

Because IPC requires kernel intervention, a process will generally first have to switch from user mode to kernel mode, shown as S 1 in Fig. 3-1. This requires changing the memory map in the MMU, as well as flushing the TLB. Within the kernel, a process context switch takes place (52 in the figure), after which the other party can be activated by switching from kernel mode to user mode again (53 in Fig. 3-1). The latter switch again requires changing the MMU map and flushing the TLB.

Instead of using processes, an application can also be constructed such that different parts are executed by separate threads. Communication between those parts

is entirely dealt with by using shared data. Thread switching can sometimes be done entirely in user space, although in other implementations, the kernel is aware of threads and schedules them. The effect can be a dramatic improvement in performance.

Finally, there is also a pure software engineering reason to use threads: many applications are simply easier to structure as a collection of cooperating threads. Think of applications that need to perform several (more or less independent) tasks. For example, in the case of a word processor, separate threads can be used for handling user input, spelling and grammar checking, document layout, index generation, etc.

Thread Implementation

Threads are often provided in the form of a thread package. Such a package contains operations to create and destroy threads as well as operations on synchronization variables such as mutexes and condition variables. There are basically two approaches to implement a thread package. The first approach is to construct a thread library that is executed entirely in user mode. The second approach is to have the kernel be aware of threads and schedule them.

A user-level thread library has a number of advantages. First, it is cheap to create and destroy threads. Because all thread administration is kept in the user's address space, the price of creating a thread is primarily determined by the cost for allocating memory to set up a thread stack. Analogously, destroying a thread mainly involves freeing memory for the stack, which is no longer used. Both operations are cheap.

A second advantage of user-level threads is that switching thread context can often be done in just a few instructions. Basically, only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched. There is no need to change memory maps, flush the TLB, do CPU accounting, and so on. Switching thread context is done when two threads need to synchronize, for example, when entering a section of shared data.

However, a major drawback of user-level threads is that invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process. As we explained, threads are particularly useful to structure large applications into parts that could be logically executed at the same time. In that case, blocking on I/O should not prevent other parts to be executed in the meantime. For such applications, user-level threads are of no help.

These problems can be mostly circumvented by implementing threads in the operating system's kernel. Unfortunately, there is a high price to pay: every thread operation (creation, deletion, synchronization, etc.), will have to be carried out by

the kernel. requiring a system call. Switching thread contexts may now become as expensive as switching process contexts. As a result, most of the performance benefits of using threads instead of processes then disappears.

A solution lies in a hybrid form of user-level and kernel-level threads, generally referred to as **lightweight** processes (LWP). An LWP runs in the context of a single (heavy-weight) process, and there can be several LWPs per process. In addition to having LWPs, a system also offers a user-level thread package. offering applications the usual operations for creating and destroying threads. In addition. the package provides facilities for thread synchronization. such as mutexes and condition variables. The important issue is that the thread package is implemented entirely in user space. In other words. all operations on threads are carried out without intervention of the kernel.

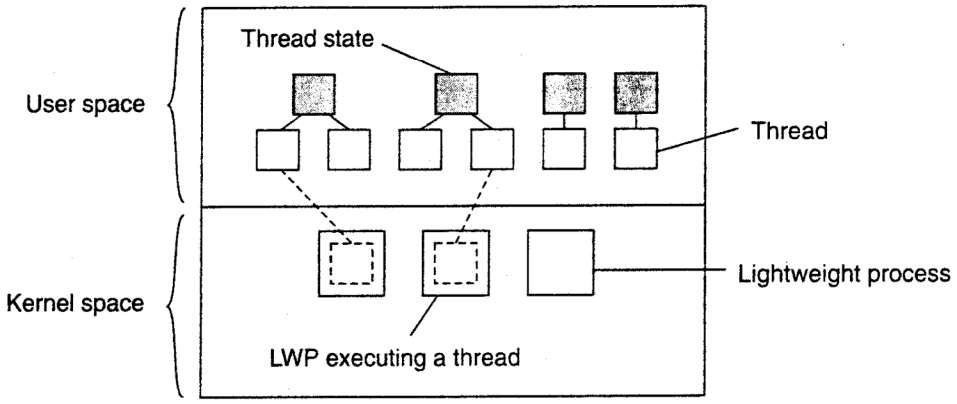

Figure 3-2. Combining kernel-level lightweight processes and user-level threads.

The thread package can be shared by multiple LWPs, as shown in Fig. 3-2. This means that each LWP can be running its own (user-level) thread. Multithreaded applications are constructed by creating threads, and subsequently assigning each thread to an LWP. Assigning a thread to an LWP is normally implicit and hidden from the programmer.

The combination of (user-level) threads and L\VPs works as follows. The thread package has a single routine to schedule the next thread. When creating an LWP (which is done by means of a system call), the LWP is given its own stack, and is instructed to execute the scheduling routine in search of a thread to execute. If there are several LWPs, then each of them executes the scheduler. The thread table, which is used to keep track of the current set of threads, is thus shared by the LWPs. Protecting this table to guarantee mutually exclusive access is done by means of mutexes that are implemented entirely in user space. In other words, synchronization between LWPs does not require any kernel support.

When an LWP finds a runnable thread, it switches context to that thread. Meanwhile, other LWPs may be looking for other runnable threads as well. If a

thread needs to block on a mutex or condition variable, it does the necessary administration and eventually calls the scheduling routine. 'When another runnable thread has been found, a context switch is made to that thread. The beauty of all this is that the LWP executing the thread need not be informed: the context switch is implemented completely in user space and appears to the LWP as normal program code.

Now let us see what happens when a thread does a blocking system call. In that case, execution changes from user mode to kernel mode. but still continues in the context of the current LWP. At the point where the current LWP can no longer continue, the operating system may decide to switch context to another LWP, which also implies that a context switch is made back to user mode. The selected LWP will simply continue where it had previously left off.

There are several advantages to using LWPs in combination with a user-level thread package. First, creating, destroying, and synchronizing threads is relatively cheap and involves no kernel intervention at all. Second, provided that a process has enough LWPs, a blocking system call will not suspend the entire process. Third, there is no need for an application to know about the LWPs. All it sees are user-level threads. Fourth, LWPs can be easily used in multiprocessing environments, by executing different LWPs on different CPUs. This multiprocessing can be hidden entirely from the application. The only drawback of lightweight processes in combination with user-level threads is that we still need to create and destroy LWPs, which is just as expensive as with kernel-level threads. However, creating and destroying LWPs needs to be done only occasionally, and is often fully controlled by the operating system.

An alternative, but similar approach to lightweight processes, is to make use of scheduler activations (Anderson et al., 1991). The most essential difference between scheduler activations and LWPs is that when a thread blocks on a system call, the kernel does an *upcall* to the thread package, effectively calling the scheduler routine to select the next runnable thread. The same procedure is repeated when a thread is unblocked. The advantage of this approach is that it saves management of LWPs by the kernel. However, the use of upcalls is considered less elegant, as it violates the structure of layered systems, in which calls only to the next lower-level layer are permitted.

3.1.2 Threads in Distributed Systems

An important property of threads is that they can provide a convenient means of allowing blocking system calls without blocking the entire process in which the thread is running. This property makes threads particularly attractive to use in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time. We illustrate this point by taking a closer look at multithreaded clients and servers, respectively.

Multithreaded Clients.

To establish a high degree of distribution transparency, distributed systems that operate in wide-area networks may need to conceal long interprocess message propagation times. The round-trip delay in a wide-area network can easily be in the order of hundreds of milliseconds. or sometimes even seconds.

The usual way to hide communication latencies is to initiate communication and immediately proceed with something else. A typical example where this happens is in Web browsers. In many cases, a Web document consists of an HTML file containing plain text along with a collection of images, icons, etc. To fetch each element of a Web document, the browser has to set up a TCPIIP connection, read the incoming data, and pass it to a display component. Setting up a connection as well as reading incoming data are inherently blocking operations. When dealing with long-haul communication, we also have the disadvantage that the time for each operation to complete may be relatively long.

A Web browser often starts with fetching the HTML page and subsequently displays it. To hide communication latencies as much as possible, some browsers start displaying data while it is still coming in. While the text is made available to the user, including the facilities for scrolling and such, the browser continues with fetching other files that make up the page, such as the images. The latter are displayed as they are brought in. The user need thus not wait until all the components of the entire page are fetched before the page is made available.

In effect, it is seen that the Web browser is doing a number of tasks simultaneously. As it turns out, developing the browser as a multithreaded client simplifies matters considerably. As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts. Each thread sets up a separate connection to the server and pulls in the data. Setting up a connection and reading data from the server can be programmed using the standard (blocking) system calls, assuming that a blocking call does not suspend the entire process. As is also illustrated in Stevens (1998), the code for each thread is the same and, above all, simple. Meanwhile, the user notices only delays in the display of images and such, but can otherwise browse through the document.

There is another important benefit to using multithreaded Web browsers in which several connections can be opened simultaneously. In the previous example, several connections were set up to the same server. If that server is heavily loaded, or just plain slow, no real performance improvements will be noticed compared to pulling in the files that make up the page strictly one after the other.

However, in many cases, Web servers have been replicated across multiple machines, where each server provides exactly the same set of Web documents. The replicated servers are located at the same site, and are known under the same name. When a request for a Web page comes in, the request is forwarded to one of the servers, often using a round-robin strategy or some other load-balancing technique (Katz et al., 1994). When using a multithreaded client, connections may

be set up to different replicas, allowing data to be transferred in parallel, effectively establishing that the entire Web document is fully displayed in a much shorter time than with a nonreplicated server. This approach is possible only if the client can handle truly parallel streams of incoming data. Threads are ideal for this purpose.

~ultithreaded Servers

Although there are important benefits to multithreaded clients, as we have seen, the main use of multithreading in distributed systems is found at the server side. Practice shows that multithreading not only simplifies server code considerably, but also makes it much easier to develop servers that exploit parallelism to attain high performance, even on uniprocessor systems. However, now that multiprocessor computers are widely available as general-purpose workstations, multithreading for parallelism is even more useful.

To understand the benefits of threads for writing server code, consider the organization of a file server that occasionally has to block waiting for the disk. The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply. One possible, and particularly popular organization is shown in Fig. 3-3. Here one thread, the **dispatcher,** reads incoming requests for a file operation. The requests are sent by clients to a well-known end point for this server. After examining the request, the server chooses an idle (i.e., blocked) **worker thread** and hands it the request.

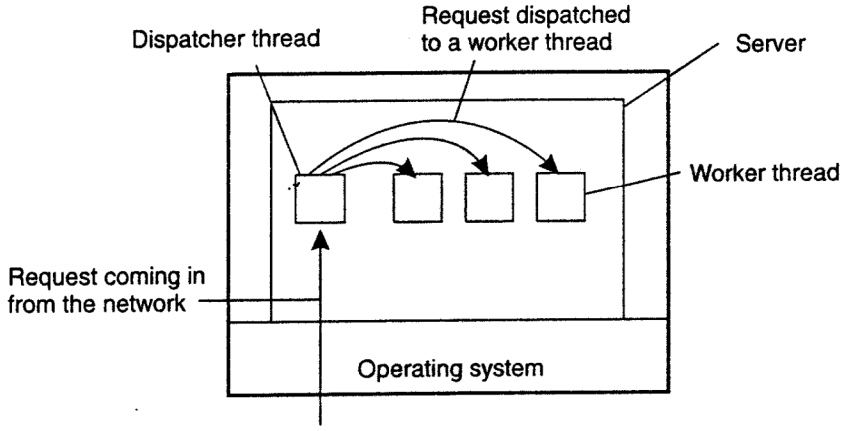

Figure 3-3. A multithreaded server organized in a dispatcher/worker model.

The worker proceeds by performing a blocking read on the *local* file system, which may cause the thread to be suspended until the data are fetched from disk. If the thread is suspended, another thread is selected to be executed. For example, the dispatcher may be selected to acquire more work. Alternatively, another worker thread can be selected that is now ready to run.

Now consider how the file server might have been written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other requests. Consequently, requests from other clients cannot be handled. In addition, if the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk. The net result is that many fewer requests/sec can be processed. Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way.

So far we have seen two possible designs: a multithreaded file server and a single-threaded file server. Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. A third possibility is to run the server as a big finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a message must be sent to the disk.

However, instead of blocking, it records the state of the current request in a table and then goes and gets the next message. The next message may either be a request for new work or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed and subsequently sent to the client. In this scheme, the server will have to make use of nonblocking calls to send and receive.

In this design, the "sequential process" model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table for every message sent and received. In effect, we are simulating threads and their stacks the hard way. The process is being operated as a finite-state machine that gets an event and then reacts to it, depending on what is in it.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Figure 3-4. Three ways to construct a server.

It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls (e.g., an RPC to talk to the disk) and still achieve parallelism. Blocking system calls make programming easier and parallelism improves performance. The single-threaded server retains the ease and simplicity of blocking system calls, but gives up some

amount of performance. The finite-state machine approach achieves high performance through parallelism, but uses nonblocking calls, thus is hard to program. These models are summarized in Fig. 3-4.

3.2 VIRTUALIZATION

Threads and processes can be seen as a way to do more things at the same time. In effect, they allow us build (pieces of) programs that appear to be executed simultaneously. On a single-processor computer, this simultaneous execution is, of course, an illusion. As there is only a single CPU, only an instruction from a single thread or process will be executed at a time. By rapidly switching between threads and processes, the illusion of parallelism is created.

This separation between having a single CPU and being able to pretend there are more can be extended to other resources as well, leading to what is known as resource virtualization. This virtualization has been applied for many decades, but has received renewed interest as (distributed) computer systems have become more commonplace and complex, leading to the situation that application software is mostly always outliving its underlying systems software and hardware. In this section, we pay some attention to the role of virtualization and discuss how it can be realized.

3.2.1 The Role of Virtualization in Distributed Systems

In practice, every (distributed) computer system offers a programming interface to higher level software, as shown in Fig. 3-5(a). There are many different types of interfaces, ranging from the basic instruction set as offered by a CPU to the vast collection of application programming interfaces that are shipped with many current middleware systems. In its essence, virtualization deals with extending or replacing an existing interface so as to mimic the behavior of another system, as shown in Fig.3-5(b). We will come to discuss technical details on virtualization shortly, but let us first concentrate on why virtualization is important for distributed systems.

One of the most important reasons for introducing virtualization in the 1970s, was to allow legacy software to run on expensive mainframe hardware. The software not only included various applications, but in fact also the operating systems they were developed for. This approach toward supporting legacy software has been successfully applied on the IBM 370 mainframes (and their successors) that offered a virtual machine to which different operating systems had been ported.

As hardware became cheaper, computers became more powerful, and the number of different operating system flavors was reducing, virtualization became less of an issue. However, matters have changed again since the late 1990s for several reasons, which we will now discuss.

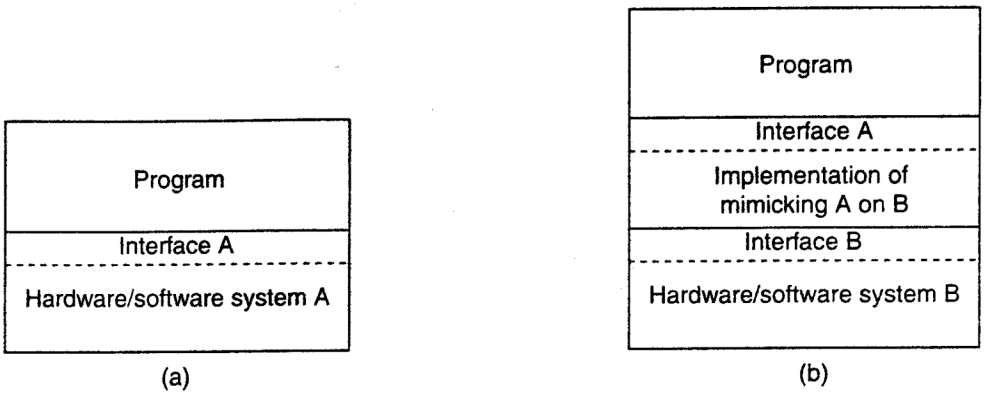

Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.

First, while hardware and low-level systems software change reasonably fast, software at higher levels of abstraction (e.g., middleware and applications), are much more stable. In other words, we are facing the situation that legacy software cannot be maintained in the same pace as the platforms it relies on. Virtualization can help here by porting the legacy interfaces to the new platforms and thus immediately opening up the latter for large classes of existing programs.

Equally important is the fact that networking has become completely pervasive. It is hard to imagine that a modern computer is not connected to a network. In practice, this connectivity requires that system administrators maintain a large and heterogeneous collection of server computers, each one running very different applications, which can be accessed by clients. At the same time the various resources should be easily accessible to these applications. Virtualization can help a lot: the diversity of platforms and machines can be reduced by essentially letting each application run on its own virtual machine, possibly including the related libraries *and* operating system, which, in turn, run on a common platform.

This last type of virtualization provides a high degree of portability and flexibility. For example, in order to realize content delivery networks that can easily support replication of dynamic content, Awadallah and Rosenblum (2002) argue that management becomes much easier if edge servers would support virtualization, allowing a complete site, including its environment to be dynamically copied. As we will discuss later, it is primarily such portability arguments that make virtualization an important mechanism for distributed systems.

3.2.2 Architectures of Virtual Machines

There are many different ways in which virtualization can be realized in practice. An overview of these various approaches is described by Smith and Nair (2005). To understand the differences in virtualization, it is important to realize

that computer systems generally offer four different types of interfaces, at four different levels:

1. An interface between the hardware and software, consisting of machine instructions that can be invoked by any program.
2. An interface between the hardware and software, consisting of machine instructions that can be invoked only by privileged programs, such as an operating system.
3. An interface consisting of system calls as offered by an operating system.
4. An interface consisting of library calls, generally forming what is known as an application programming interface (API). In many cases, the aforementioned system calls are hidden by an API.

These different types are shown in Fig. 3-6. The essence of virtualization is to mimic the behavior of these interfaces.

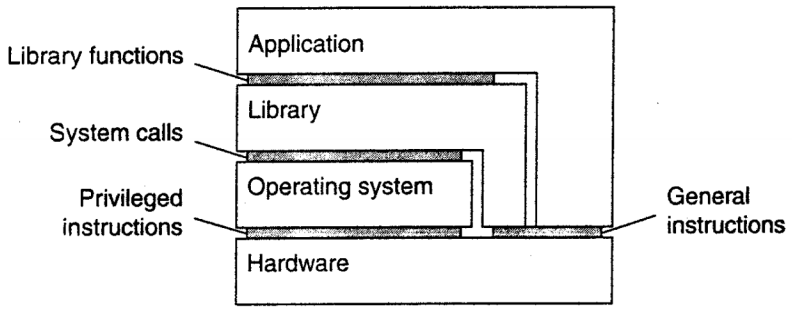

Figure 3-6. Various interfaces offered by computer systems.

Virtualization can take place in two different ways. First, we can build a runtime system that essentially provides an abstract instruction set that is to be used for executing applications. Instructions can be interpreted (as is the case for the Java runtime environment), but could also be emulated as is done for running Windows applications on UNIX platforms. Note that in the latter case, the emulator will also have to mimic the behavior of system calls, which has proven to be generally far from trivial. This type of virtualization leads to what Smith and Nair (2005) call a process virtual machine, stressing that virtualization is done essentially only for a single process.

An alternative approach toward virtualization is to provide a system that is essentially implemented as a layer completely shielding the original hardware, but offering the complete instruction set of that same (or other hardware) as an interface. Crucial is the fact that this interface can be offered *simultaneously* to different programs. As a result, it is now possible to have multiple, and different

operating systems run independently and concurrently on the same platform. The layer is generally referred to as a **virtual machine monitor** (VMM). Typical examples of this approach are VMware (Sugerman et al., 2001) and Xen (Barham et at, 2003). These two different approaches are shown in Fig. 3-7.

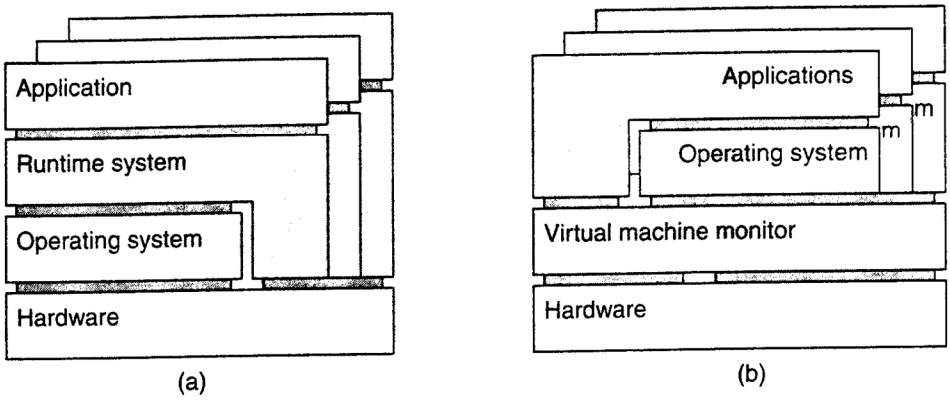

Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations. (b) A virtual machine monitor. with multiple instances of (applications, operating system) combinations.

As argued by Rosenblum and Garfinkel (2005), VMMs will become increasingly important in the context of reliability and security for (distributed) systems. As they allow for the isolation of a complete application and its environment, a failure caused by an error or security attack need no longer affect a complete machine. In addition, as we also mentioned before, portability is greatly improved as VMMs provide a further decoupling between hardware and software, allowing a complete environment to be moved from one machine to another.

3.3 CLIENTS

In the previous chapters we discussed the client-server modeL the roles of clients and servers, and the ways they interact. Let us now take a closer look at the anatomy of clients and servers, respectively. We start in this section with a discussion of clients. Servers are discussed in the next section.

3.3.1 Networked User Interfaces

A major task of client machines is to provide the means for users to interact with remote servers. There are roughly two ways in which this interaction can be supported. First, for each remote service the client machine will have a separate counterpart that can contact the service over the network. A typical example is an agenda running on a user's PDA that needs to synchronize with a remote, possibly

shared agenda. In this case, an application-level protocol will handle the synchronization, as shown in Fig. 3-8(a).

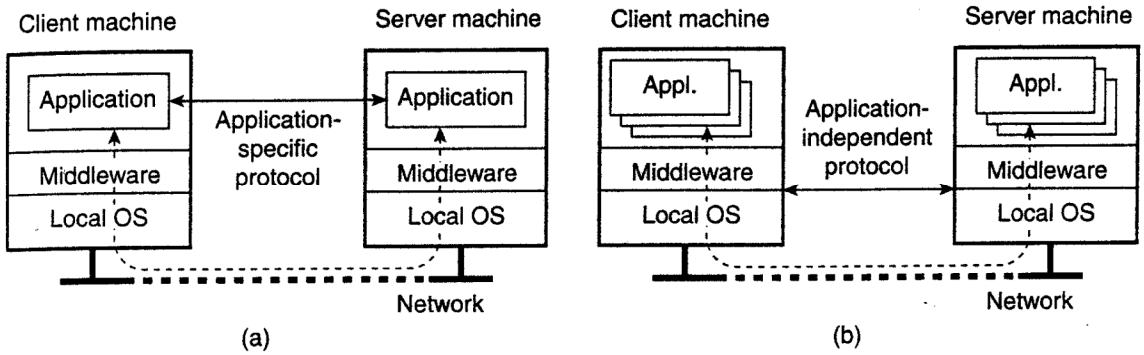

Figure 3-8. (a) A networked application with its own protocol. (b) A general solution to allow access to remote applications.

A second solution is to provide direct access to remote services by only offering a convenient user interface. Effectively, this means that the client machine is used only as a terminal with no need for local storage, leading to an application-neutral solution as shown in Fig. 3-8(b). In the case of networked user interfaces, everything is processed and stored at the server. This thin-client approach is receiving more attention as Internet connectivity increases, and hand-held devices are becoming more sophisticated. As we argued in the previous chapter, thin-client solutions are also popular as they ease the task of system management. Let us take a look at how networked user interfaces can be supported.

Example: The X Window System

Perhaps one of the oldest and still widely-used networked user interfaces is the X Window system. The X Window System, generally referred to simply as X, is used to control bit-mapped terminals, which include a monitor, keyboard, and a pointing device such as a mouse. In a sense, X can be viewed as that part of an operating system that controls the terminal. The heart of the system is formed by what we shall call the X kernel. It contains all the terminal-specific device drivers, and as such, is generally highly hardware dependent.

The X kernel offers a relatively low-level interface for controlling the screen, but also for capturing events from the keyboard and mouse. This interface is made available to applications as a library called *Xlib*. This general organization is shown in Fig. 3-9.

The interesting aspect of X is that the X kernel and the X applications need not necessarily reside on the same machine. In particular, X provides the X protocol, which is an application-level communication protocol by which an instance of *Xlib* can exchange data and events with the X kernel. For example, *Xlib* can send

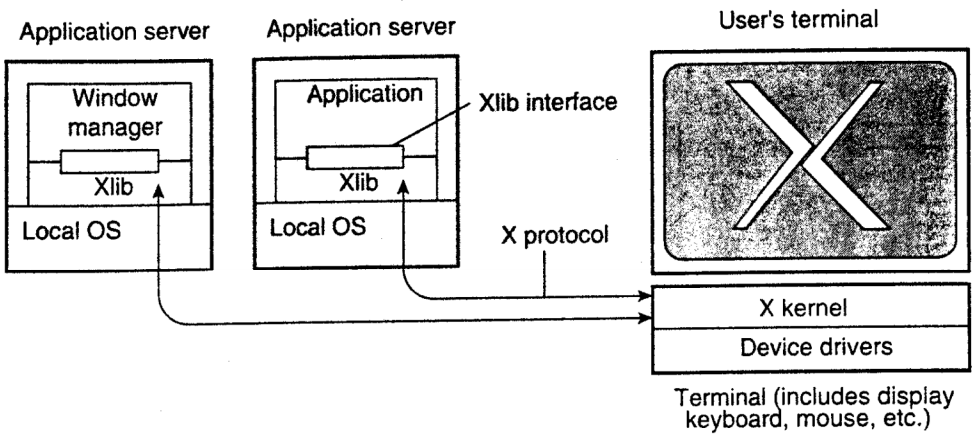

Figure 3-9. The basic organization of the X Window System.

requests to the X kernel for creating or killing a window, setting colors, and defining the type of cursor to display, among many other requests. In turn, the X kernel will react to local events such as keyboard and mouse input by sending event packets back to *Xlib*.

Several applications can communicate at the same time with the X kernel. There is one specific application that is given special rights, known as the **window manager.** This application can dictate the "look and feel" of the display as it appears to the user. For example, the window manager can prescribe how each window is decorated with extra buttons, how windows are to be placed on the display, and so. Other applications will have to adhere to these rules.

It is interesting to note how the X window system actually fits into client-server computing. From what we have described so far, it should be clear that the X kernel receives requests to manipulate the display. It gets these requests from (possibly remote) applications. In this sense, the X kernel acts as a server, while the applications play the role of clients. This terminology has been adopted by X, and although strictly speaking is correct, it can easily lead to confusion.

Thin-Client Network Computing

Obviously, applications manipulate a display using the specific display commands as offered by X. These commands are generally sent over the network where they are subsequently executed by the X kernel. By its nature, applications written for X should preferably separate application logic from user-interface commands. Unfortunately, this is often not the case. As reported by Lai and Nieh (2002), it turns out that much of the application logic and user interaction are tightly coupled, meaning that an application will send many requests to the X kernel for which it will expect a response before being able to make a next step. This

synchronous behavior may adversely affect performance when operating over a wide-area network with long latencies.

There are several solutions to this problem. One is to re-engineer the implementation of the X protocol, as is done with NX (Pinzari, 2003). An important part of this work concentrates on bandwidth reduction by compressing X messages. First, messages are considered to consist of a fixed part, which is treated as an identifier, and a variable part. In many cases, multiple messages will have the same identifier in which case they will often contain similar data. This property can be used to send only the differences between messages having the same identifier.

Both the sending and receiving side maintain a local cache of which the entries can be looked up using the identifier of a message. When a message is sent, it is first looked up in the local cache. If found, this means that a previous message with the same identifier but possibly different data had been sent. In that case, differential encoding is used to send only the differences between the two. At the receiving side, the message is also looked up in the local cache, after which decoding through the differences can take place. In the cache miss, standard compression techniques are used, which generally already leads to factor four improvement in bandwidth. Overall, this technique has reported bandwidth reductions up to a factor 1000, which allows X to also run through low-bandwidth links of only 9600 kbps.

An important side effect of caching messages is that the sender and receiver have shared information on what the current status of the display is. For example, the application can request geometric information on various objects by simply requesting lookups in the local cache. Having this shared information alone already reduces the number of messages required to keep the application and the display synchronized.

Despite these improvements, X still requires having a display server running. This may be asking a lot, especially if the display is something as simple as a cell phone. One solution to keeping the software at the display very simple is to let all the processing take place at the application side. Effectively, this means that the entire display is controlled up to the pixel level at the application side. Changes in the bitmap are then sent over the network to the display, where they are immediately transferred to the local frame buffer.

This approach requires sophisticated compression techniques in order to prevent bandwidth availability to become a problem. For example, consider displaying a video stream at a rate of 30 frames per second on a 320 x 240 screen. Such a screen size is common for many PDAs. If each pixel is encoded by 24 bits, then without compression we would need a bandwidth of approximately 53 Mbps. Compression is clearly needed in such a case, and many techniques are currently being deployed. Note, however, that compression requires decompression at the receiver, which, in turn, may be computationally expensive without hardware support. Hardware support can be provided, but this raises the devices cost.

The drawback of sending raw pixel data in comparison to higher-level protocols such as X is that it is impossible to make any use of application semantics, as these are effectively lost at that level. Baratto et al. (2005) propose a different technique. In their solution, referred to as THINC, they provide a few high-level display commands that operate at the level ofthe video device drivers. These commands are thus device dependent, more powerful than raw pixel operations, but less powerful compared to what a protocol such as X offers. The result is that display servers can be much simpler, which is good for CPU usage, while at the same time application-dependent optimizations can be used to reduce bandwidth and synchronization.

In THINC, display requests from the application are intercepted and translated into the lower level commands. By intercepting application requests, THINe can make use of application semantics to decide what combination of lower level commands can be used best. Translated commands are not immediately sent out to the display, but are instead queued. By batching several commands it is possible to aggregate display commands into a single one, leading to fewer messages. For example, when a new command for drawing in a particular region of the screen effectively overwrites what a previous (and still queued) command would have established, the latter need not be sent out to the display. Finally, instead of letting the display ask for refreshments, THINC always pushes updates as they come available. This push approach saves latency as there is no need for an update request to be sent out by the display.

As it turns out, the approach followed by THINC provides better overall performance, although very much in line with that shown by NX. Details on performance comparison can be found in Baratto et al. (2005).

Compound Documents

Modem user interfaces do a lot more than systems such as X or its simple applications. In particular, many user interfaces allow applications to share a single graphical window, and to use that window to exchange data through user actions. Additional actions that can be performed by the user include what are generally called drag-and-drop operations, and in-place editing, respectively.

A typical example of drag-and-drop functionality is moving an icon representing a file *A* to an icon representing a trash can, resulting in the file being deleted. In this case, the user interface will need to do more than just arrange icons on the display: it will have to pass the name of the file *A* to the application associated with the trash can as soon as *A's* icon has been moved above that of the trash can application. Other examples easily come to mind.

In-place editing can best be illustrated by means of a document containing text and graphics. Imagine that the document is being displayed within a standard word processor. As soon as the user places the mouse above an image, the user interface passes that information to a drawing program to allow the user to modify

the image. For example, the user may have rotated the image, which may effect the placement of the image in the document. The user interface therefore finds out what the new height and width of the image are, and passes this information to the word processor. The latter, in turn, can then automatically update the page layout of the document.

The key idea behind these user interfaces is the notion of a compound document, which can be defined as a collection of documents, possibly of very different kinds (like text, images, spreadsheets, etc.), which are seamlessly integrated at the user-interface level. A user interface that can handle compound documents hides the fact that different applications operate on different parts of the document. To the user, all parts are integrated in a seamless way. When changing one part affects other parts, the user interface can take appropriate measures, for example, by notifying the relevant applications.

Analogous to the situation described for the X Window System, the applications associated with a compound document do not have to execute on the client's machine. However, it should be clear that user interfaces that support compound documents may have to do a lot more processing than those that do not.

3.3.2 Client-Side Software for Distribution Transparency

Client software comprises more than just user interfaces. In many cases, parts of the processing and data level in a client-server application are executed on the client side as well. A special class is formed by embedded client software, such as for automatic teller machines (ATMs), cash registers, barcode readers, TV set-top boxes, etc. In these cases, the user interface is a relatively small part of the client software, in contrast to the local processing and communication facilities.

Besides the user interface and other application-related software, client software comprises components for achieving distribution transparency. Ideally, a client should not be aware that it is communicating with remote processes. In contrast, distribution is often less transparent to servers for reasons of performance and correctness. For example, in Chap. 6 we will show that replicated servers sometimes need to communicate in order to establish that operations are performed in a specific order at each replica.

Access transparency is generally handled through the generation of a client stub from an interface definition of what the server has to offer. The stub provides the same interface as available at the server, but hides the possible differences in machine architectures, as well as the actual communication.

There are different ways to handle location, migration, and relocation transparency. Using a convenient naming system is crucial, as we shall also see in the next chapter. In many cases, cooperation with client-side software is also important. For example, when a client is already bound to a server, the client can be directly informed when the server changes location. In this case, the client's middleware can hide the server's current geographical location from the user, and

also transparently rebind to the server if necessary. At worst, the client's application may notice a temporary loss of performance.

In a similar way, many distributed systems implement replication transparency by means of client-side solutions. For example, imagine a distributed system with replicated servers, Such replication can be achieved by forwarding a request to each replica, as shown in Fig. 3-10. Client-side software can transparently collect all responses and pass a single response to the client application.

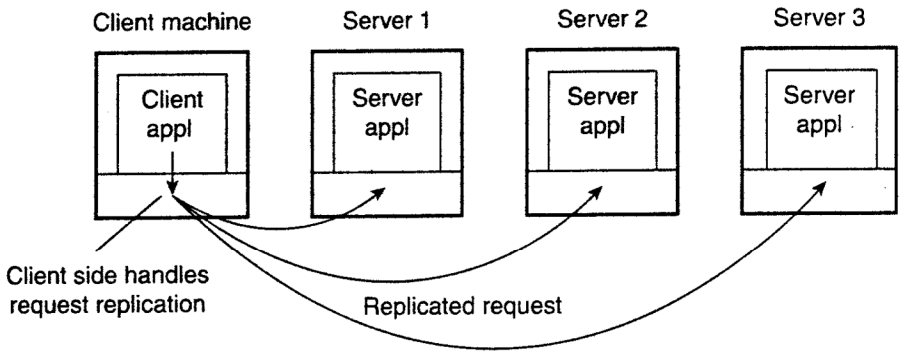

Figure 3-10. Transparent replication of a server using a client-side solution.

Finally, consider failure transparency. Masking communication failures with a server is typically done through client middleware. For example, client middleware can be configured to repeatedly attempt to connect to a server, or perhaps try another server after several attempts. There are even situations in which the client middleware returns data it had cached during a previous session, as is sometimes done by Web browsers that fail to connect to a server.

Concurrency transparency can be handled through special intermediate servers, notably transaction monitors, and requires less support from client software. Likewise, persistence transparency is often completely handled at the server.

3.4 SERVERS

Let us now take a closer look at the organization of servers. In the following pages, we first concentrate on a number of general design issues for servers, to be followed by a discussion of server clusters.

3.4.1 General Design Issues

A server is a process implementing a specific service on behalf of a collection of clients. In essence, each server is organized in the same way: it waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.

There are several ways to organize servers. In the case of an iterative server, the server itself handles the request and, if necessary, returns a response to the requesting client. A concurrent server does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request. A multithreaded server is an example of a concurrent server. An alternative implementation of a concurrent server is to fork a new process for each new incoming request. This approach is followed in many UNIX systems. The thread or process that handles the request is responsible for returning a response to the requesting client.

Another issue is where clients contact a server. In all cases, clients send requests to an end point, also called a port, at the machine where the server is running. Each server listens to a specific end point. How do clients know the end point of a service? One approach is to globally assign end points for well-known services. For example, servers that handle Internet FTP requests always listen to TCP port 21. Likewise, an HTTP server for the World Wide Web will always listen to TCP port 80. These end points have been assigned by the Internet Assigned Numbers Authority (lANA), and are documented in Reynolds and Postel (1994). With assigned end points, the client only needs to find the network address of the machine where the server is running. As we explain in the next chapter, name services can be used for that purpose.

There are many services that do not require a preassigned end point. For example, a time-of-day server may use an end point that is dynamically assigned to it 9Y its local operating system. In that case, a client will first have to look up the end point. One solution is to have a special daemon running on each machine that runs servers. The daemon keeps track of the current end point of each service implemented by a co-located server. The daemon itself listens to a well-known end point. A client will first contact the daemon, request the end point, and then c~mtact the specific server, as shown in Fig. 3-11(a).

It is common to associate an end point with a specific service. However, actually implementing each service by means of a separate server may be a waste of resources. For example, in a typical UNIX system, it is common to have lots of servers running simultaneously, with most of them passively waiting until a client request comes in. Instead of having to keep track of so many passive processes, it is often more efficient to have a single superserver listening to each end point associated with a specific service, as shown in Fig. 3-1l(b). This is the approach taken, for example, with the *inetd* daemon in UNIX. *Inetd* listens to a number of well-known ports for Internet services. When a request comes in, the daemon forks a process to take further care of the request. That process will exit after it is finished.

Another issue that needs to be taken into account when designing a server is whether and how a server can be interrupted. For example, consider a user who has just decided to upload a huge file to an FTP server. Then, suddenly realizing that it is the wrong file, he wants to interrupt the server to cancel further data

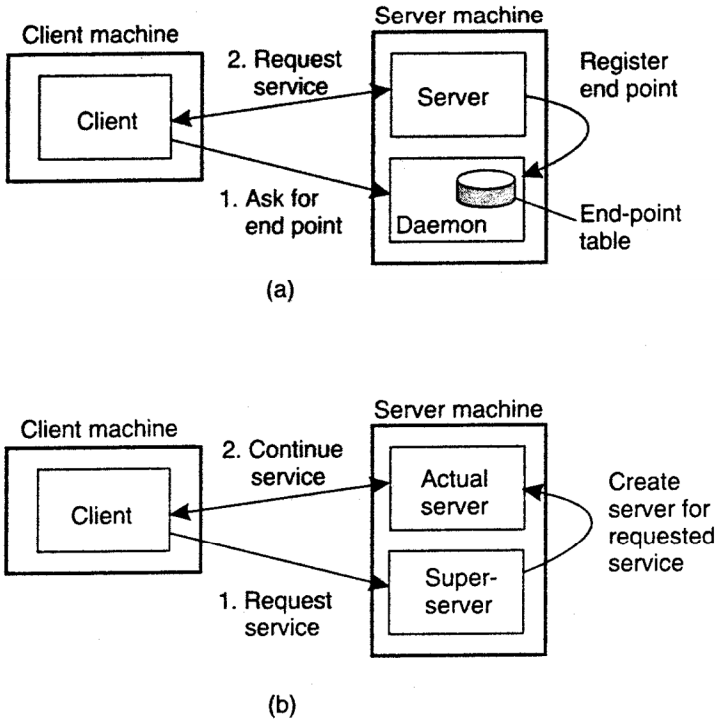

Figure 3-11. (a) Client-to-server binding using a daemon. (b) Client-to-server binding using a superserver..

transmission. There are several ways to do this. One approach that works only too well in the current Internet (and is sometimes the only alternative) is for the user to abruptly exit the client application (which will automatically break the connection to the server), immediately restart it, and pretend nothing happened. The server will eventually tear down the old connection, thinking the client has probably crashed.

A much better approach for handling communication interrupts is to develop the client and server such that it is possible to send **out-of-band** data, which is data that is to be processed by the server before any other data from that client. One solution is to let the server listen to a separate control end point to which the client sends out-of-band data, while at the same time listening (with a lower priority) to the end point through which the normal data passes. Another solution is to send out-of-band data across the same connection through which the client is sending the original request. In TCP, for example, it is possible to transmit urgent data. When urgent data are received at the server, the latter is interrupted (e.g., through a signal in UNIX systems), after which it can inspect the data and handle them accordingly.

A final, important design issue, is whether or not the server is stateless. A stateless server does not keep information on the state of its clients, and can change its own state without having to inform any client (Birman, 2005). A Web

server, for example, is stateless. It merely responds to incoming HTTP requests, which can be either for uploading a file to the server or (most often) for fetching a file. When the request has been processed, the Web server forgets the client completely. Likewise, the collection of files that a Web server manages (possibly in cooperation with a file server), can be changed without clients having to be informed.

Note that in many stateless designs, the server actually does maintain information on its clients, but crucial is the fact that if this information is lost, it will not lead to a disruption of the service offered by the server. For example, a Web server generally logs all client requests. This information is useful, for example, to decide whether certain documents should be replicated, and where they should be replicated to. Clearly, there is no penalty other than perhaps in the form of suboptimal performance if the log is lost.

A particular form of a stateless design is where the server maintains what is known as soft state. In this case, the server promises to maintain state on behalf of the client, but only for a limited time. After that time has expired, the server falls back to default behavior, thereby discarding any information it kept on account of the associated client. An example of this type of state is a server promising to keep a client informed about updates, but only for a limited time. After that, the client is required to poll the server for updates. Soft-state approaches originate from protocol design in computer networks, but can be equally applied to server design (Clark, 1989; and Lui et al., 2004).

In contrast, a stateful server generally maintains persistent information on its clients. This means that the information needs to be explicitly deleted by the server. A typical example is a file server that allows a client to keep a local copy of a file, even for performing update operations. Such a server would maintain a table containing *(client, file)* entries. Such a table allows the server to keep track of which client currently has the update permissions on which file, and thus possibly also the most recent version of that file.

This approach can improve the performance of read and write operations as perceived by the client. Performance improvement over stateless servers is often an important benefit of stateful designs. However, the example also illustrates the major drawback of stateful servers. If the server crashes, it has to recover its table of *(client, file)* entries, or otherwise it cannot guarantee that it has processed the most recent updates on a file. In general, a stateful server needs to recover its entire state as it was just before the crash. As we discuss in Chap. 8, enabling recovery can introduce considerable complexity. In a stateless design, no special measures need to be taken at all for a crashed server to recover. It simply starts running again, and waits for client requests to come in.

Ling et al. (2004) argue that one should actually make a distinction between (temporary) session state and permanent state. The example above is typical for session state: it is associated with a series of operations by a single user and should be maintained for a some time, but not indefinitely. As it turns out, session

state is often maintained in three-tiered client-server architectures, where the application server actually needs to access a database server through a series of queries before being able to respond to the requesting client. The issue here is that no real harm is done if session state is lost, provided that the client can simply reissue the original request. This observation allows for simpler and less reliable storage of state.

What remains for permanent state is typically information maintained in databases, such as customer information, keys associated with purchased software, etc. However, for most distributed systems, maintaining session state already implies a stateful design requiring special measures when failures do happen and making explicit assumptions about the durability of state stored at the server. We will return to these matters extensively when discussing fault tolerance.

When designing a server, the choice for a stateless or stateful design should not affect the services provided by the server. For example, if files have to be opened before they can be read from, or written to, then a stateless server should one way or the other mimic this behavior. A common solution, which we discuss in more detail in Chap. 11. is that the server responds to a read or write request by first opening the referred file, then does the actual read or write operation, and immediately closes the file again.

In other cases, a server may want to keep a record on a client's behavior so that it can more effectively respond to its requests. For example, Web servers sometimes offer the possibility to immediately direct a client to his favorite pages. This approach is possible only if the server has history information on that client. When the server cannot maintain state, a common solution is then to let the client send along additional information on its previous accesses. In the case of the Web, this information is often transparently stored by the client's browser in what is called a cookie, which is a small piece of data containing client-specific information that is of interest to the server. Cookies are never executed by a browser; they are merely stored.

The first time a client accesses a server, the latter sends a cookie along with the requested Web pages back to the browser, after which the browser safely tucks the cookie away. Each subsequent time the client accesses the server, its cookie for that server is sent along with the request. Although in principle, this approach works fine, the fact that cookies are sent back for safekeeping by the browser is often hidden entirely from users. So much for privacy. Unlike most of grandma's cookies, these cookies should stay where they are baked.

3.4.2 Server Clusters

In Chap. 1 we briefly discussed cluster computing as one of the many appearances of distributed systems. We now take a closer look at the organization of server clusters, along with the salient design issues.

General Organization

Simply put, a server cluster is nothing else but a collection of machines connected through a network, where each machine runs one or more servers. The server clusters that we consider here, are the ones in which the machines are connected through a local-area network, often offering high bandwidth and low latency.

In most cases, a server cluster is logically organized into three tiers, as shown in Fig. 3-12. The first tier consists of a (logical) switch through which client requests are routed. Such a switch can vary widely. For example, transport-layer switches accept incoming TCP connection requests and pass requests on to one of servers in the cluster, as we discuss below. A completely different example is a Web server that accepts incoming HTTP requests, but that partly passes requests to application servers for further processing only to later collect results and return an HTTP response.

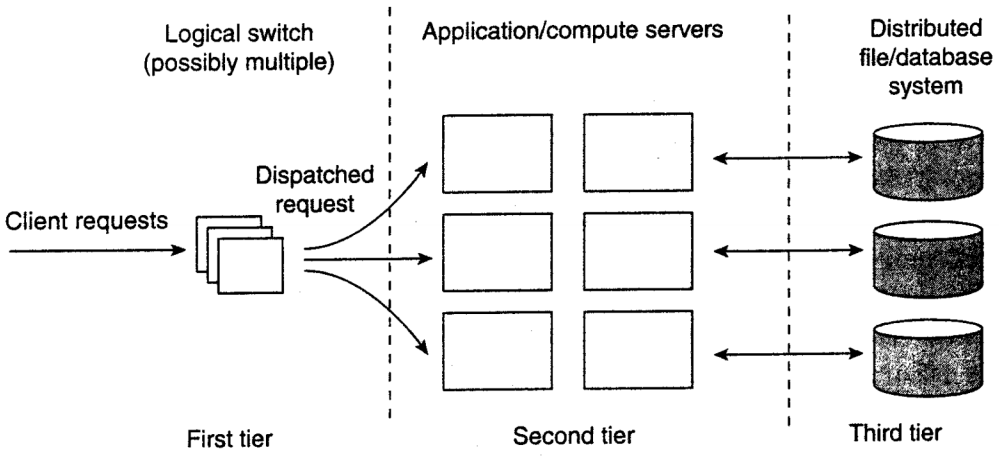

Figure 3-12. The general organization of a three-tiered server cluster.

As in any multitiered client-server architecture, many server clusters also contain servers dedicated to application processing. In cluster computing, these are typically servers running on high-performance hardware dedicated to delivering compute power. However, in the case of enterprise server clusters, it may be the case that applications need only run on relatively low-end machines, as the required compute power is not the bottleneck, but access to storage is.

This brings us the third tier, which consists of data-processing servers, notably file and database servers. Again, depending on the usage of the server cluster, these servers may be running an specialized machines, configured for high-speed disk access and having large server-side data caches.

Of course, not all server clusters will follow this strict separation. It is frequently the case that each machine is equipped with its own local storage, often

integrating application and data processing in a single server leading to a two-tiered architecture. For example, when dealing with streaming media by means of a server cluster, it is common to deploy a two-tiered system architecture, where each machine acts as a dedicated media server (Steinmetz and Nahrstedt, 2004).

When a server cluster offers multiple services, it may happen that different machines run different application servers. As a consequence, the switch will have to be able to distinguish services or otherwise it cannot forward requests to the proper machines. As it turns out, many second-tier machines run only a single application. This limitation comes from dependencies on available software and hardware, but also that different applications are often managed by different administrators. The latter do not like to interfere with each other's machines.

As a consequence, we may find that certain machines are temporarily idle, while others are receiving an overload of requests. What would be useful is to temporarily migrate services to idle machines. A solution proposed in Awadallah and Rosenblum (2004), is to use virtual machines allowing a relative easy migration of code to real machines. We will return to code migration later in this chapter.

Let us take a closer look at the first tier, consisting of the switch. An important design goal for server clusters is to hide the fact that there are multiple servers. In other words, client applications running on remote machines should have no need to know anything about the internal organization of the cluster. This access transparency is invariably offered by means of a single access point, in turn implemented through some kind of hardware switch such as a dedicated machine.

The switch forms the entry point for the server cluster, offering a single network address. For scalability and availability, a server cluster may have multiple access points, where each access point is then realized by a separate dedicated machine. We consider only the case of a single access point.

A standard way of accessing a server cluster is to set up a TCP connection over which application-level requests are then sent as part of a session. A session ends by tearing down the connection. In the case of transport-layer switches, the switch accepts incoming TCP connection requests, and hands off such connections to one of the servers (Hunt et al, 1997; and Pai et al., 1998). The principle working of what is commonly known as TCP handoff is shown in Fig. 3-13.

When the switch receives a TCP connection request, it subsequently identifies the best server for handling that request, and forwards the request packet to that server. The server, in turn, will send an acknowledgment back to the requesting client, but inserting the switch's IP address as the source field of the header of the IP packet carrying the TCP segment. Note that this spoofing is necessary for the client to continue executing the TCP protocol: it is expecting an answer back from the switch, not from some arbitrary server it is has never heard of before. Clearly, a TCP-handoff implementation requires operating-system level modifications.

It can already be seen that the switch can play an important role in distributing the load among the various servers. By deciding where to forward a request to, the

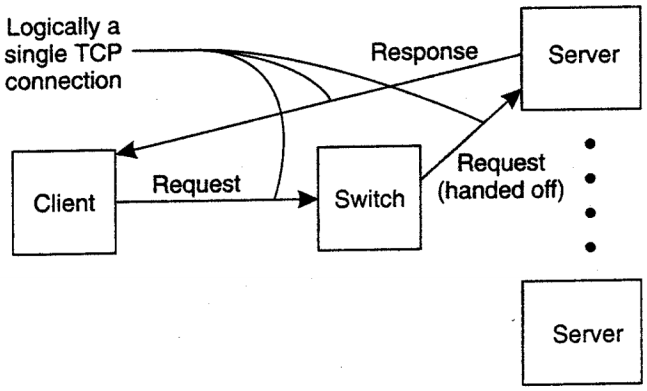

Figure 3-13. The principle ofTCP handoff.

switch also decides which server is to handle further processing of the request: The simplest load-balancing policy that the switch can follow is round robin: each time it picks the next server from its list to forward a request to.

More advanced server selection criteria can be deployed as well. For example, assume multiple services are offered by the server cluster. If the switch can distinguish those services when a request comes in, it can then take informed decisions on where to forward the request to. This server selection can still take place at the transport level, provided services are distinguished by means of a port number. One step further is to have the switch actually inspect the payload of the incoming request. This method can be applied only if it is known what that payload can look like. For example, in the case of Web servers, the switch can eventually expect an HTTP request, based on which it can then decide who is to process it. We will return to such content-aware request distribution when we discuss Web-based systems in Chap. 12.

Distributed Servers

The server clusters discussed so far are generally rather statically configured. In these clusters, there is often an separate administration machine that keeps track of available servers, and passes this information to other machines as appropriate, such as the switch.

As we mentioned, most server clusters offer a single access point. When that point fails, the cluster becomes unavailable. To eliminate this potential problem, several access points can be provided, of which the addresses are made publicly available. For example, the Domain Name System (DNS) can return several addresses, all belonging to the same host name. This approach still requires clients to make several attempts if one of the addresses fails. Moreover, this does not solve the problem of requiring static access points.

Having stability, like a long-living access point, is a desirable feature from a client's and a server's perspective. On the other hand, it also desirable to have a high degree of flexibility in configuring a server cluster, including the switch. This observation has lead to a design of a distributed server which effectively is nothing but a possibly dynamically changing set of machines, with also possibly varying access points, but which nevertheless- appears to the outside world as a single. powerful machine. The design of such a distributed server is given in Szymaniak et al. (2005). We describe it briefly here.

The basic idea behind a distributed server is that clients benefit from a robust, high-performing, stable server. These properties can often be provided by high-end mainframes, of which some have an acclaimed mean time between failure of more than 40 years. However, by grouping simpler machines transparently into a cluster, and not relying on the availability of a single machine, it may be possible to achieve a better degree of stability than by each component individually. For example, such a cluster could be dynamically configured from end-user machines, as in the case of a collaborative distributed system.

Let us concentrate on how a stable access point can be achieved in such a system. The main idea is to make use of available networking services, notably mobility support for IP version 6 (MIPv6). In MIPv6, a mobile node is assumed to have a home network where it normally resides and for which it has an associated stable address, known as its home address (HoA). This home network has a special router attached, known as the home agent, which will take care of traffic to the mobile node when it is away. To this end, when a mobile node attaches to a foreign network, it will receive a temporary care-of address (CoA) where it can be reached. This care-of address is reported to the node's home agent who will then see to it that all traffic is forwarded to the mobile node. Note that applications communicating with the mobile node will only see the address associated with the node's home network. They will never see the care-of address.

This principle can be used to offer a stable address of a distributed server. In this case, a single unique contact address is initially assigned to the server cluster. The contact address will be the server's life-time address to be used in all communication with the outside world. At any time, one node in the distributed server will operate as an access point using that contact address, but this role can easily be taken over by another node. What happens is that the access point records its own address as the care-of address at the home agent associated with the distributed server. At that point, all traffic will be directed to the access point, who will then take care in distributing requests among the currently participating nodes. If the access point fails, a simple fail-over mechanism comes into place by which another access point reports a new care-of address.

This simple configuration would make the home agent as well as the access point a potential bottleneck as all traffic would flow through these two machines. This situation can be avoided by using an MIPv6 feature known as *route optimization*. Route optimization works as follows. Whenever a mobile node with home

address *HA* reports its current care-of address, say *CA*, the horne agent can forward *CA* to a client. The latter will then locally store the pair *(HA, CA)*. From that moment on, communication will be directly forwarded to *CA*. Although the application at the client side can still use the horne address, the underlying support software for MIPv6 will translate that address to *CA* and use that instead.

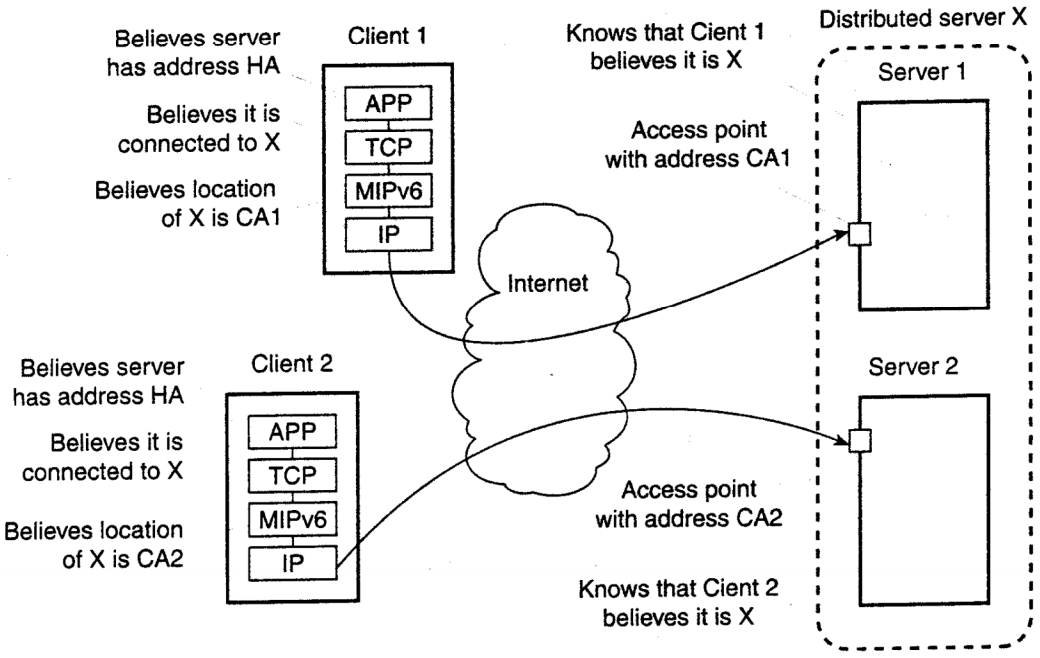

Figure 3-14. Route optimization in a distributed server.

Route optimization can be used to make different clients believe they are communicating with a single server, where, in fact, each client is communicating with a different member node of the distributed server, as shown in Fig. 3-14. To this end, when an access point of a distributed server forwards a request from client C1 to, say node S1 (with address *CA 1)'* it passes enough information to S1 to let it initiate the route optimization procedure by which eventually the client is made to believe that the care-of address is *CA* 1. This will allow C1 to store the pair *(HA, CA* 1)' During this procedure, the access point (as well as the horne agent) tunnel most of the traffic between C1 and S1. This will prevent the horne agent from believing that the care-of address has changed, so that it will continue to communicate with the access point.

Of course, while this route optimization procedure is taking place, requests from other clients may still corne in. These remain in a pending state at the access point until they can be forwarded. The request from another client C2 may then be forwarded to member node S2 (with address *CA* 2), allowing the latter to let client

C_Z store the pair (HA, CA_2). As a result, different clients will be directly communicating with different members of the distributed server, where each client application still has the illusion that this server has address HA. The home agent continues to communicate with the access point talking to the contact address.

3.4.3 Managing Server Clusters

A server cluster should appear to the outside world as a single computer, as is indeed often the case. However, when it comes to managing a cluster, the situation changes dramatically. Several attempts have been made to ease the management of server clusters as we discuss next.

Common Approaches

By far the most common approach to managing a server cluster is to extend the traditional managing functions of a single computer to that of a cluster. In its most primitive form, this means that an administrator can log into a node from a remote client and execute local managing commands to monitor, install, and change components.

Somewhat more advanced is to hide the fact that you need to login into a node and instead provide an interface at an administration machine that allows to collect information from one or more servers, upgrade components, add and remove nodes, etc. The main advantage of the latter approach is that collective operations, which operate on a group of servers, can be more easily provided. This type of managing server clusters is widely applied in practice, exemplified by management software such as Cluster Systems Management from IBM (Hochstetler and Beringer, 2004).

However, as soon as clusters grow beyond several tens of nodes, this type of management is not the way to go. Many data centers need to manage thousands of servers, organized into many clusters but all operating collaboratively. Doing this by means of centralized administration servers is simply out of the question. Moreover, it can be easily seen that very large clusters need continuous repair management (including upgrades). To simplify matters, if p is the probability that a server is currently faulty, and we assume that faults are independent, then for a cluster of N servers to operate without a single server being faulty is $(1-p)^N$. With $p=0.001$ and $N=1000$, there is only a 36 percent chance that all the servers are correctly functioning.

As it turns out, support for very large server clusters is almost always ad hoc. There are various rules of thumb that should be considered (Brewer, 2001), but there is no systematic approach to dealing with massive systems management. Cluster management is still very much in its infancy, although it can be expected that the self-managing solutions as discussed in the previous chapter will eventually find their way in this area, after more experience with them has been gained.

Example: PlanetLab

Let us now take a closer look at a somewhat unusual cluster server. PlanetLab is a collaborative distributed system in which different organizations each donate one or more computers, adding up to a total of hundreds of nodes. Together, these computers form a l-tier server cluster, where access, processing, and storage can all take place on each node individually. Management of PlanetLab is by necessity almost entirely distributed. Before we explain its basic principles, let us first describe the main architectural features (Peterson et al., 2005).

In PlanetLab, an organization donates one or more nodes, where each node is easiest thought of as just a single computer, although it could also be itself a cluster of machines. Each node is organized as shown in Fig. 3-15. There are two important components (Bavier et al., 2004). The first one is the virtual machine monitor (VMM), which is an enhanced Linux operating system. The enhancements mainly comprise adjustments for supporting the second component, namely vservers. A (Linux) vserver can best be thought of as a separate environment in which a group of processes run. Processes from different vservers are *completely* independent. They cannot directly share any resources such as files, main memory, and network connections as is normally the case with processes running on top of an operating systems. Instead, a vserver provides an environment consisting of its own collection of software packages, programs, and networking facilities. For example, a vserver may provide an environment in which a process will notice that it can make use of Python 1.5.2 in combination with an older Apache Web server, say *httpd* 1.3.1. In contrast, another vserver may support the latest versions of Python and *httpd.* In this sense, calling a vserver a "server" is a bit of a misnomer as it really only isolates groups of processes from each other. We return to vservers briefly below.

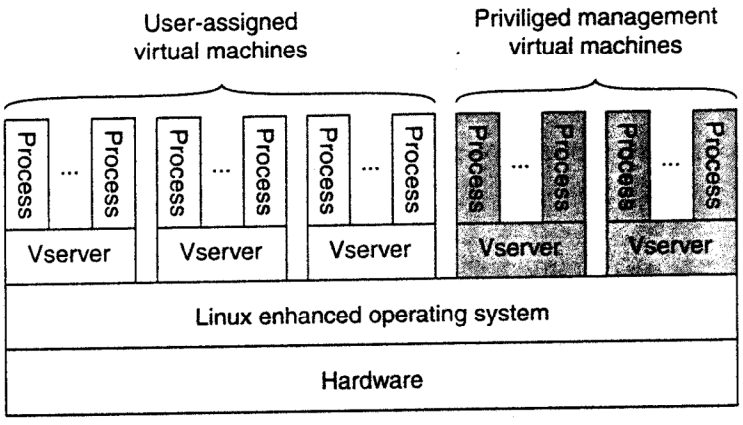

Figure 3-15. The basic organization of a PlanetLab node.

The Linux VMM ensures that vservers are separated: processes in different vservers are executed concurrently and independently, each making use only of

the software packages and programs available in their own environment. The isolation between processes in different vservers is strict. For example, two processes in different vservers may have the same user ill, but this does not imply that they stem from the same user. This separation considerably eases supporting users from different organizations that want to use PlanetLab as, for example, a testbed to experiment with completely different distributed systems and applications.

To support such experimentations, PlanetLab introduces the notion of a slice, which is a set of vservers, each vserver running on a different node. A slice can thus be thought of as a virtual server cluster, implemented by means of a collection of virtual machines. The virtual machines in PlanetLab run on top of the Linux operating system, which has been extended with a number of kernel modules

There are several issues that make management of PlanetLab a special problem. Three salient ones are:

1. Nodes belong to different organizations. Each organization should be allowed to specify who is allowed to run applications on their nodes, and restrict resource usage appropriately.
2. There are various monitoring tools available, but they all assume a very specific combination of hardware and software. Moreover, they are all tailored to be used within a single organization.
3. Programs from different slices but running on the same node should not interfere with each other. This problem is similar to process independence in operating systems.

Let us take a look at each of these issues in more detail.

Central to managing PlanetLab resources is the node manager. Each node has such a manager, implemented by means of a separate vserver, whose only task is to create other vservers on the node it manages and to control resource allocation. The node manager does not make any policy decisions; it is merely a mechanism to provide the essential ingredients to get a program running on a given node.

Keeping track of resources is done by means of a resource specification, or *rspee* for short. An *rspee* specifies a time interval during which certain resources have been allocated. Resources include disk space, file descriptors, inbound and outbound network bandwidth, transport-level end points, main memory, and CPU usage. An *rspee* is identified through a globally unique 128-bit identifier known as a resource capability *(reap).* Given an *reap,* the node manager can look up the associated *rspee* in a local table.

Resources are bound to slices. In other words, in order to make use of resources, it is necessary to create a slice. Each slice is associated with a service provider, which can best be seen as an entity having an account on PlanetLab.

Every slice can then be identified by a *(principal.sid, , slice.uag)* pair, where the *principal.iid* identifies the provider and *slice.stag* is an identifier chosen by the provider.

To create a new slice, each node will run a slice creation service (SCS), which, in tum, can contact the node manager requesting it to create a vserver and to allocate resources. The node manager itself cannot be contacted directly over a network, allowing it to concentrate only on local resource management. In tum, the SCS will not accept slice-creation requests from just anybody. Only speeific slice authorities are eligible for requesting the creation of a slice. Each slice authority will have access rights to a collection of nodes. The simplest model is that there is only a single slice authority that is allowed to request slice creation on all nodes.

To complete the picture, a service provider will contact a slice authority and request it to create a slice across a collection of nodes. The service provider will be known to the slice authority, for example, because it has been previously authenticated and subsequently registered as a PlanetLab user. In practice, PlanetLab users contact a slice authority by means of a Web-based service. Further details can be found in Chun and Spalink (2003).

What this procedure reveals is that managing PlanetLab is done through intermediaries. One important class of such intermediaries is formed by slice authorities. Such authorities have obtained credentials at nodes to create slides. Obtaining these credentials has been achieved out-of-band, essentially by contacting system administrators at various sites. Obviously, this is a time-consuming process which not be carried out by end users (or, in PlanetLab terminology; service providers).

Besides slice authorities, there are also management authorities. Where a slice authority concentrates only on managing slices, a management authority is responsible for keeping an eye on nodes. In particular, it ensures that the nodes under its regime run the basic PlanetLab software and abide to the rules set out by PlanetLab. Service providers trust that a management authority provides nodes that will behave properly.

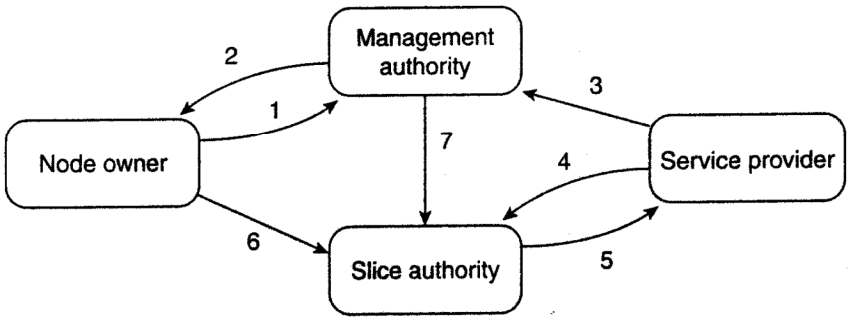

Figure 3-16. The management relationships between various PlanetLab entities.

This organization leads to the management structure shown in Fig. 3-16. described in terms of trust relationships in Peterson et at (2005). The relations are as follows:

1. A node owner puts its node under the regime of a management authority, possibly restricting usage where appropriate.
2. A management authority provides the necessary software to add a node to PlanetLab.
3. A service provider registers itself with a management authority. trusting it to provide well-behaving nodes.
4. A service provider contacts a slice authority to create a slice on a collection of nodes.
5. The slice authority needs to authenticate the service provider.
6. A node owner provides a slice creation service for a slice authority to create slices. It essentially delegates resource management to the slice authority.
7. A management authority delegates the creation of slices to a slice authority.

These relationships cover the problem of delegating nodes in a controlled way such that a node owner can rely on a decent and secure management. The second issue that needs to be handled is monitoring. What is needed is a unified approach to allow users to see how well their programs are behaving within a specific slice.

PlanetLab follows a simple approach. Every node is equipped with a collection of sensors, each sensor being capable of reporting information such as CPU usage, disk activity, and so on. Sensors can be arbitrarily complex, but the important issue is that,they always report information on a per-node basis. This information is made available by means of a Web server: every sensor is accessible through simple HTTP requests (Bavier et at, 2004).

Admittedly, this approach to monitoring is still rather primitive, but it should be seen as a basis for advanced 'monitoring schemes. For example, there is, in principle, no reason why Astrolabe, which we discussed in Chap. 2, cannot be used for aggregated sensor readings across multiple nodes.

Finally, to come to our third management issue, namely the protection of programs against each other, PlanetLab uses Linux virtual servers (called vservers) to isolate slices. As mentioned, the main idea of a vserver is to run applications in there own environment, which includes all files that are normally shared across a single machine. Such a separation can be achieved relatively easy by means of the UNIX chroot command, which effectively changes the root of the file system from where applications will look for files. Only the superuser can execute chroot.

Of course, more is needed. Linux virtual servers not only separate the file system, but also normally shared information on processes, network addresses, memory usage, and so on. As a consequence, a physical machine is actually partitioned into multiple units, each unit corresponding to a full-fledged Linux environment, isolated from the other parts. An overview of Linux virtual servers can be found in Potzl et al. (2005).

3.5 CODE MIGRATION

So far, we have been mainly concerned with distributed systems in which communication is limited to passing data. However, there are situations in which passing programs, sometimes even while they are being executed, simplifies the design of a distributed system. In this section, we take a detailed look at what code migration actually is. We start by considering different approaches to code migration, followed by a discussion on how to deal with the local resources that a migrating program uses. A particularly hard problem is migrating code in heterogeneous systems, which is also discussed.

3.5.1 Approaches to Code Migration

Before taking a look at the different forms of code migration, let us first consider why it may be useful to migrate code.

Reasons for Migrating Code

Traditionally, code migration in distributed systems took place in the form of process migration in which an entire process was moved from one machine to another (Milojicic et al., 2000). Moving a running process to a different machine is a costly and intricate task, and there had better be a good reason for doing so. That reason has always been performance. The basic idea is that overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines. Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well.

Load distribution algorithms by which decisions are made concerning the allocation and redistribution of tasks with respect to a set of processors, play an important role in compute-intensive systems. However, in many modem distributed systems, optimizing computing capacity is less an issue than, for example, trying to minimize communication. Moreover, due to the heterogeneity of the underlying platforms and computer networks, performance improvement through code migration is often based on qualitative reasoning instead of mathematical models.

Consider, as an example, a client-server system in which the server manages a huge database. If a client application needs to perform many database operations

involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. Otherwise, the network may be swamped with the transfer of data from the server to the client. In this case, code migration is based on the assumption that it generally makes sense to process data close to where those data reside.

This same reason can be used for migrating parts of the server to the client. For example, in many interactive database applications, clients need to fill in forms that are subsequently translated into a series of database operations. Processing the form at the client side, and sending only the completed form to the server, can sometimes avoid that a relatively large number of small messages need to cross the network. The result is that the client perceives better performance, while at the same time the server spends less time on form processing and communication.

Support for code migration can also help improve performance by exploiting parallelism, but without the usual intricacies related to parallel programming. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program, called a mobile agent, that moves from site to site. By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speed-up compared to using just a single program instance.

Besides improving performance, there are other reasons for supporting code migration as well. The most important one is that of flexibility. The traditional approach to building distributed applications is to partition the application into different parts, and decide in advance where each part should be executed. This approach, for example, has led to the different multitiered client-server applications discussed in Chap. 2.

However, if code can move between different machines, it becomes possible to dynamically configure distributed systems. For example, suppose a server implements a standardized interface to a file system. To allow remote clients to access the file system, the server makes use of a proprietary protocol. Normally, the client-side implementation of the file system interface, which is based on that protocol, would need to be linked with the client application. This approach requires that the software be readily available to the client at the time the client application is being developed.

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server. At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. This principle is shown in Fig. 3-17. This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine. Different solutions are discussed below and in later chapters.

The important advantage of this model of dynamically downloading client-side software is that clients need not have all the software preinstalled to talk to

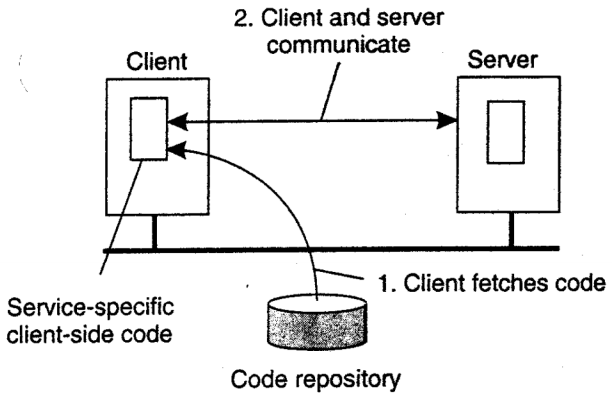

Figure 3-17. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

servers. Instead, the software can be moved in as necessary, and likewise, discarded when no longer needed. Another advantage is that as long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like. Changes will not affect existing client applications that rely on the server. There are, of course, also disadvantages. The most serious one, which we discuss in Chap. 9, has to do with security. Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard disk and does not send the juiciest parts to heaven-knows-who may not always be such a good idea.

Models for Code Migration

Although code migration suggests that we move only code between machines, the term actually covers a much richer area. Traditionally, communication in distributed systems is concerned with exchanging data between processes. Code migration in the broadest sense deals with moving programs between machines, with the intention to have those programs be executed at the target. In some cases, as in process migration, the execution status of a program, pending signals, and other parts of the environment must be moved as well.

To get a better understanding of the different models for code migration, we use a framework described in Fuggetta et al. (1998). In this framework, a process consists of three segments. The *code segment* is the part that contains the set of instructions that make up the program that is being executed. The *resource segment* contains references to external resources needed by the process, such as files, printers, devices, other processes, and so on. Finally, an *execution segment* is used to store the current execution state of a process, consisting of private data, the stack, and, of course, the program counter.

The bare minimum for code migration is to provide only weak mobility. In this model, it is possible to transfer only the code segment, along with perhaps some initialization data. A characteristic feature of weak mobility is that a transferred program is always started from one of several predefined starting positions. This is what happens, for example, with Java applets, which always start execution from the beginning. The benefit of this approach is its simplicity. Weak mobility requires only that the target machine can execute that code, which essentially boils down to making the code portable. We return to these matters when discussing migration in heterogeneous systems.

In contrast to weak mobility, in systems that support strong mobility the execution segment can be transferred as well. The characteristic feature of strong mobility is that a running process can be stopped, subsequently moved to another machine, and then resume execution where it left off. Clearly, strong mobility is much more general than weak mobility, but also much harder to implement.

Irrespective of whether mobility is weak or strong, a further distinction can be made between sender-initiated and receiver-initiated migration. In sender-initiated migration, migration is initiated at the machine where the code currently resides or is being executed. Typically, sender-initiated migration is done when uploading programs to a compute server. Another example is sending a search program across the Internet to a Web database server to perform the queries at that server. In receiver-initiated migration, the initiative for code migration is taken by the target machine. Java applets are an example of this approach.

Receiver-initiated migration is simpler than sender-initiated migration. In many cases, code migration occurs between a client and a server, where the client takes the initiative for migration. Securely uploading code to a server, as is done in sender-initiated migration, often requires that the client has previously been registered and authenticated at that server. In other words, the server is required to know all its clients, the reason being is that the client will presumably want access to the server's resources such as its disk. Protecting such resources is essential. In contrast, downloading code as in the receiver-initiated case, can often be done anonymously. Moreover, the server is generally not interested in the client's resources. Instead, code migration to the client is done only for improving client-side performance. To that end, only a limited number of resources need to be protected, such as memory and network connections. We return to secure code migration extensively in Chap. 9.

In the case of weak mobility, it also makes a difference if the migrated code is executed by the target process, or whether a separate process is started. For example, Java applets are simply downloaded by a Web browser and are executed in the browser's address space. The benefit of this approach is that there is no need to start a separate process, thereby avoiding communication at the target machine. The main drawback is that the target process needs to be protected against malicious or inadvertent code executions. A simple solution is to let the operating system take care of that by creating a separate process to execute the migrated code.

Note that this solution does not solve the resource-access problems mentioned above. They still have to be dealt with.

Instead of moving a running process, also referred to as process migration, strong mobility can also be supported by remote cloning. In contrast to process migration, cloning yields an exact copy of the original process, but now running on a different machine. The cloned process is executed in parallel to the original process. In UNIX systems, remote cloning takes place by forking off a child process and letting that child continue on a remote machine. The benefit of cloning is that the model closely resembles the one that is already used in many applications. The only difference is that the cloned process is executed on a different machine. In this sense, migration by cloning is a simple way to improve distribution transparency.

The various alternatives for code migration are summarized in Fig. 3-18.

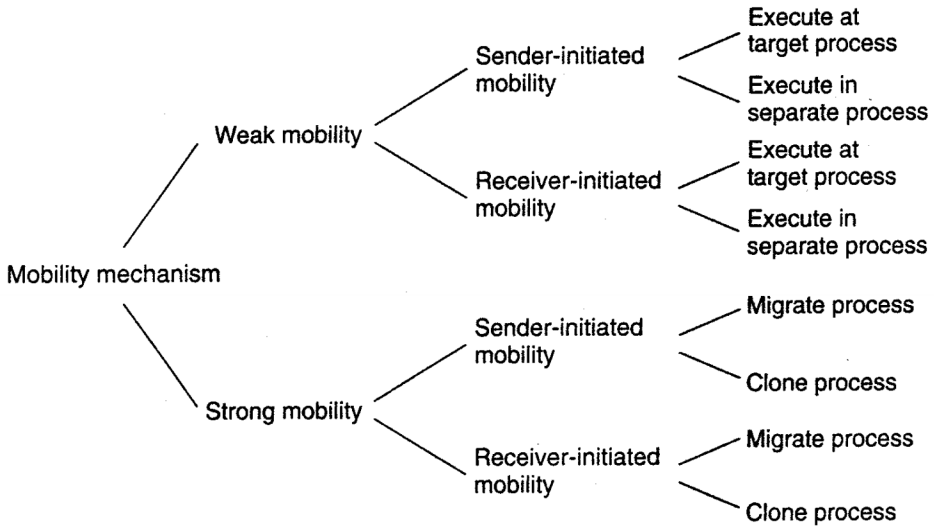

Figure 3-18. Alternatives for code migration.

3.5.2 Migration and Local Resources

So far, only the migration of the code and execution segment has been taken into account. The resource segment requires some special attention. What often makes code migration so difficult is that the resource segment cannot always be simply transferred along with the other segments without being changed. For example, suppose a process holds a reference to a specific TCP port through which it was communicating with other (remote) processes. Such a reference is held in its resource segment. When the process moves to another location, it will have to give up the port and request a new one at the destination. In other cases, transferring a reference need not be a problem. For example, a reference to a file by

means of an absolute URL will remain valid irrespective of the machine where the process that holds the URL resides.

To understand the implications that code migration has on the resource segment, Fuggetta et al. (1998) distinguish three types of process-to-resource bindings. The strongest binding is when a process refers to a resource by its identifier. In that case, the process requires precisely the referenced resource, and nothing else. An example of such a binding by identifier is when a process uses a VRL to refer to a specific Web site or when it refers to an FrP server by means of that server's Internet address. In the same line of reasoning, references to local communication end points also lead to a binding by identifier.

A weaker form of process-to-resource binding is when only the value of a resource is needed. In that case, the execution of the process would not be affected if another resource would provide that same value. A typical example of binding by value is when a program relies on standard libraries, such as those for programming in C or Java. Such libraries should always be locally available, but their exact location in the local file system may differ between sites. Not the specific files, but their content is important for the proper execution of the process.

Finally, the weakest form of binding is when a process indicates it needs only a resource of a specific type. This binding by type is exemplified by references to local devices, such as monitors, printers, and so on.

When migrating code, we often need to change the references to resources, but cannot affect the kind of process-to-resource binding. If, and exactly how a reference should be changed, depends on whether that resource can be moved along with the code to the target machine. More specifically, we need to consider the resource-to-machine bindings, and distinguish the following cases. Unattached resources can be easily moved between different machines, and are typically (data) files associated only with the program that is to be migrated. In contrast, moving or copying a fastened resource may be possible, but only at relatively high costs. Typical examples of fastened resources are local databases and complete Web sites. Although such resources are, in theory, not dependent on their current machine, it is often infeasible to move them to another environment. Finally, fixed resources are intimately bound to a specific machine or environment and cannot be moved. Fixed resources are often local devices. Another example of a fixed resource is a local communication end point.

Combining three types of process-to-resource bindings, and three types of resource-to-machine bindings, leads to nine combinations that we need to consider when migrating code. These nine combinations are shown in Fig. 3-19.

Let us first consider the possibilities when a process is bound to a resource by identifier. When the resource is unattached, it is generally best to move it along with the migrating code. However, when the resource is shared by other processes, an alternative is to establish a global reference, that is, a reference that can cross machine boundaries. An example of such a reference is a URL. When the resource is fastened or fixed, the best solution is also to create a global reference.

Process-to-resource binding	Resource-to-machine binding: Unattached	Fastened	Fixed
By identifier	MV (or GR)	GR (or MV)	GR
By value	CP (or MV,GR)	GR (or CP)	GR
By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR Establish a global systemwide reference
MV Move the resource
CP Copy the value of the resource
RB Rebind process to locally-available resource

Figure 3-19. Actions to be taken with respect to the references to local resources when migrating code to another machine.

It is important to realize that establishing a global reference may be more than just making use of URLs, and that the use of such a reference is sometimes prohibitively expensive. Consider, for example, a program that generates high-quality images for a dedicated multimedia workstation. Fabricating high-quality images in real time is a compute-intensive task, for which reason the program may be moved to a high-performance compute server. Establishing a global reference to the multimedia workstation means setting up a communication path between the compute server and the workstation. In addition, there is significant processing involved at both the server and the workstation to meet the bandwidth requirements of transferring the images. The net result may be that moving the program to the compute server is not such a good idea, only because the cost of the global reference is too high.

Another example of where establishing a global reference is not always that easy is when migrating a process that is making use of a local communication end point. In that case, we are dealing with a fixed resource to which the process is bound by the identifier. There are basically two solutions. One solution is to let the process set up a connection to the source machine after it has migrated and install a separate process at the source machine that simply forwards all incoming messages. The main drawback of this approach is that whenever the source machine malfunctions, communication with the migrated process may fail. The alternative solution is to have all processes that communicated with the migrating process, change *their* global reference, and send messages to the new communication end point at the target machine.

The situation is different when dealing with bindings by value. Consider first a fixed resource. The combination of a fixed resource and binding by value occurs, for example, when a process assumes that memory can be shared between processes. Establishing a global reference in this case would mean that we need to implement a distributed form of shared memory. In many cases, this is not really a viable or efficient solution.

Fastened resources that are referred to by their value, are typically runtime libraries. Normally, copies of such resources are readily available on the target machine, or should otherwise be copied before code migration takes place. Establishing a global reference is a better alternative when huge amounts of data are to be copied, as may be the case with dictionaries and thesauruses in text processing systems.

The easiest case is when dealing with unattached resources. The best solution is to copy (or move) the resource to the new destination, unless it is shared by a number of processes. In the latter case, establishing a global reference is the only option.

The last case deals with bindings by type. Irrespective of the resource-to-machine binding. the obvious solution is to rebind the process to a locally available resource of the same type. Only when such a resource is not available, will we need to copy or move the original one to the new destination, or establish a global reference.

3.5.3 Migration in Heterogeneous Systems

So far, we have tacitly assumed that the migrated code can be easily executed at the target machine. This assumption is in order when dealing with homogeneous systems. In general, however, distributed systems are constructed on a heterogeneous collection of platforms, each having their own operating system and machine architecture. Migration in such systems requires that each platform is supported, that is, that the code segment can be executed on each platform. Also, we need to ensure that the execution segment can be properly represented at each platform.

The problems coming from heterogeneity are in many respects the same as those of portability. Not surprisingly, solutions are also very similar. For example, at the end of the 1970s, a simple solution to alleviate many of the problems of porting Pascal to different machines was to generate machine-independent intermediate code for an abstract virtual machine (Barron, 1981). That machine, of course, would need to be implemented on many platforms, but it would then allow Pascal programs to be run anywhere. Although this simple idea was widely used for some years, it never really caught on as the general solution to portability problems for other languages, notably C.

About 25 years later, code migration in heterogeneous systems is being attacked by scripting languages and highly portable languages such as Java. In essence, these solutions adopt the same approach as was done for porting Pascal. All such solutions have in common that they rely on a (process) virtual machine that either directly interprets source code (as in the case of scripting languages), or otherwise interprets intermediate code generated by a compiler (as in Java). Being in the right place at the right time is also important for language developers.

Recent developments have started to weaken the dependency on programming languages. In particular, solutions have been proposed not only to migrate processes, but to migrate entire computing environments. The basic idea is to compartmentalize the overall environment and to provide processes in the same part their own view on their computing environment.

If the compartmentalization is done properly, it becomes possible to decouple a part from the underlying system and actually migrate it to another machine. In this way, migration would actually provide a form of strong mobility for processes, as they can then be moved at any point during their execution, and continue where they left off when migration completes. Moreover, many of the intricacies related to migrating processes while they have bindings to local resources may be solved, as these bindings are in many cases simply preserved. The local resources, namely, are often part of the environment that is being migrated.

There are several reasons for wanting to migrate entire environments, but perhaps the most important one is that it allows continuation of operation while a machine needs to be shutdown. For example, in a server cluster, the systems administrator may decide to shut down or replace a machine, but will not have to stop all its running processes. Instead, it can temporarily freeze an environment, move it to another machine (where it sits next to other, existing environments), and simply unfreeze it again. Clearly, this is an extremely powerful way to manage long-running compute environments and their processes.

Let us consider one specific example of migrating virtual machines, as discussed in Clark et al. (2005). In this case, the authors concentrated on real-time migration of a virtualized operating system, typically something that would be convenient in a cluster of servers where a tight coupling is achieved through a single, shared local-area network. Under these circumstances, migration involves two major problems: migrating the entire memory image and migrating bindings to local resources.

As to the first problem, there are, in principle, three ways to handle migration (which can be combined):

1. Pushing memory pages to the new machine and resending the ones that are later modified during the migration process.
2. Stopping the current virtual machine; migrate memory, and start the new virtual machine.
3. Letting the new virtual machine pull in new pages as needed, that is, let processes start on the new virtual machine immediately and copy memory pages on demand.

The second option may lead to unacceptable downtime if the migrating virtual machine is running a live service, that is, one that offers continuous service. On the other hand, a pure on-demand approach as represented by the third option may

extensively prolong the migration period, but may also lead to poor performance because it takes a long time before the working set of the migrated processes has been moved to the new machine.

As an alternative, Clark et al. (2005) propose to use a pre-copy approach which combines the first option, along with a brief stop-and-copy phase as represented by the second option. As it turns out, this combination can lead to service downtimes of 200 ms or less.

Concerning local resources, matters are simplified when dealing only with a cluster server. First, because there is a single network, the only thing that needs to be done is to announce the new network-to-MAC address binding, so that clients can contact the migrated processes at the correct network interface. Finally, if it can be assumed that storage is provided as a separate tier (like we showed in Fig. 3-12), then migrating binding to files is similarly simple.

The overall effect is that, instead of migrating processes, we now actually see that an entire operating system can be moved between machines.

3.6 SUMMARY

Processes play a fundamental role in distributed systems as they form a basis for communication between different machines. An important issue is how processes are internally organized and, in particular, whether or not they support multiple threads of control. Threads in distributed systems are particularly useful to continue using the CPU when a blocking I/O operation is performed. In this way, it becomes possible to build highly-efficient servers that run multiple threads in parallel, of which several may be blocking to wait until disk I/O or network communication completes.

Organizing a distributed application in terms of clients and servers has proven to be useful. Client processes generally implement user interfaces, which may range from very simple displays to advanced interfaces that can handle compound documents. Client software is furthermore aimed at achieving distribution transparency by hiding details concerning the communication with servers, where those servers are currently located, and whether or not servers are replicated. In addition, client software is partly responsible for hiding failures and recovery from failures.

Servers are often more intricate than clients, but are nevertheless subject to only a relatively few design issues. For example, servers can either be iterative or concurrent, implement one or more services, and can be stateless or stateful. Other design issues deal with addressing services and mechanisms to interrupt a server after a service request has been issued and is possibly already being processed.

Special attention needs to be paid when organizing servers into a cluster. A common objective is hide the internals of a cluster from the outside world. This

means that the organization of the cluster should be shielded from applications. To this end, most clusters use a single entry point that can hand off messages to servers in the cluster. A challenging problem is to transparently replace this single entry point by a fully distributed solution.

An important topic for distributed systems is the migration of code between different machines. Two important reasons to support code migration are increasing performance and flexibility. When communication is expensive, we can sometimes reduce communication by shipping computations from the server to the client, and let the client do as much local processing as possible. Flexibility is increased if a client can dynamically download software needed to communicate with a specific server. The downloaded software can be specifically targeted to that server, without forcing the client to have it preinstalled.

Code migration brings along problems related to usage of local resources for which it is required that either resources are migrated as well, new bindings to local resources at the target machine are established, or for which systemwide network references are used. Another problem is that code migration requires that we take heterogeneity into account. Current practice indicates that the best solution to handle heterogeneity is to use virtual machines. These can take either the form of process virtual machines as in the case of, for example, Java, or through using virtual machine monitors that effectively allow the migration of a collection of processes along with their underlying operating system.

PROBLEMS

1. In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server. It takes 15 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in a cache in main memory. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. How many requests/sec can the server handle if it is single threaded? If it is multithreaded?
2. Would it make sense to limit the number of threads in a server process?
3. In the text, we described a multithreaded tile server, showing why it is better than a single-threaded server and a finite-state machine server. Are there any circumstances in which a single-threaded server might be better? Give an example.
4. Statically associating only a single thread with a lightweight process is not such a good idea. Why not?
5. Having only a single lightweight process per process is also not such a good idea. Why not?
6. Describe a simple scheme in which there are as many lightweight processes as there are runnable threads.

7. X designates a user's terminal as hosting the server, while the application is referred to as the client. Does this make sense?
8. The X protocol suffers from scalability problems. How can these problems be tackled?
9. Proxies can support replication transparency by invoking each replica, as explained in the text. Can (the server side of) an application be subject to a replicated calls?
10. Constructing a concurrent server by spawning a process has some advantages and disadvantages compared to multithreaded servers. Mention a few.
11. Sketch the design of a multithreaded server that supports multiple protocols using sockets as its transport-level interface to the underlying operating system.
12. How can we prevent an application from circumventing a window manager, and thus being able to completely mess up a screen?
13. Is a server that maintains a TCP/IP connection to a client stateful or stateless?
14. Imagine a Web server that maintains a table in which client IP addresses are mapped to the most recently accessed Web pages. When a client connects to the server, the server looks up the client in its table, and if found, returns the registered page. Is this server stateful or stateless?
15. Strong mobility in UNIX systems could be supported by allowing a process to fork a child on a remote machine. Explain how this would work.
16. In Fig. 3-18 it is suggested that strong mobility cannot be combined with executing migrated code in a target process. Give a counterexample.
17. Consider a process P that requires access to file F which is locally available on the machine where P is currently running. When P moves to another machine, it still requires access to F. If the file-to-machine binding is fixed, how could the systemwide reference to F be implemented?
18. Describe in detail how TCP packets flow in the case of TCP handoff, along with the information on source and destination addresses in the various headers.