27. For this exercise you are to implement a simple client-server system using RPC. The server offers one procedure, next, which takes an integer as input and returns its successor as output. Write a stub procedure called next for use on the client side. Its job is to send the parameter to the server using UDP and wait for the response, timing out if the response takes too long. The server procedure should listen on a known port, accept requests, carry them out, and send back the results.

In the preceding chapter, we concentrated on communication in distributed systems. Communication takes place between processes, and in this chapter, we take a closer look at how the different types of processes play a 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 turn 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.

As we argued in Chap. 1, 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. In addition, we consider general-purpose object servers, which form the basic means of implementing distributed objects.

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.

Our last subject deals with an upcoming phenomenon, namely that of software agents. In contrast to the somewhat asymmetric client-server model, multiaxial systems roughly consist of a collection of equally important agents that collectively attempt to reach a common goal. A software agent is yet another type of process and may come in different forms. Taking a distributed-system perspective, what an agent is, and how agents collaborate, is discussed in the last section.

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, 1998) 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 a different program. To keep track of these virtual processors, the operating system has a process table, containing entries to store CPU register values, a process memory maps, open files, accounting information, privileges, etc. A process is the entity that is currently being executed as a program in execution, that is, a program that is currently being run. An important issue 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 processes. In other words, the fact that multiple processes may be concurrently executing, 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 creates a common data area, the operating system must create a complete copy of the data, independent address space. Allocation can mean initializing memory segments by zeroing a data segment, copying the associated program into a text segment, and setting up a stack for temporary data. Likewise, switching 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.

A thread is very similar to a process in the sense that it can also be seen as the execution of a (part of a) program on a virtual processor. 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, the performance of a multithreaded application need hardly ever be worse than that of its single-threaded counterpart. In fact, as with many applications, multithreading leads to an increase in performance gain. Second, because threads are not automatically protected against each other the way processes are, development of multithreaded applications requires some 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.

The most important benefit comes from the fact that in a single-threaded process, whenever a blocking system call is executed, the process as a whole is blocked. To illustrate, consider an application such as spreadsheet program, and assume that a user continuously and interactively wants to change values. An important property of a spreadsheet program is that it maintains the functional 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 propagation 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.

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, 1992). 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.](image)

Because IPC requires kernel intervention, a process will generally first have to switch from user mode to kernel mode, shown as S1 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 (S2 in the figure), after which the other party can be activated by switching from kernel mode to user mode again (S3 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 generally 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 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 (see also Sec. 1.4). 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 LWPs 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, providing 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 multithreading 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 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, 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 a file 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 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 to fetch other files that make up the page, such as the images. The latter are fetched as they are brought in. The user need 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 standard blocking system calls, assuming that a blocking call does not suspend the thread.

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, and are known under the same name. When a request for a Web page comes in, the request is forwarded to one of the replicated servers. 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 replicated servers. The replicated servers are located at the same site, and are known under the same name.

Multithreaded 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 endpoint 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.](image)

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 selected, 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 could be 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, the server examines it. If it can be satisfied from the cache, fine, in, the one and only thread examines it. 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 the 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.

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls</td>
</tr>
</tbody>
</table>

Figure 3.4. Three ways to construct a server.

It should now be clear what threads 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 of blocking system calls, but gives up performance. The finite-state machine approach achieves high performance through parallelism, but uses non-blocking calls, thus is hard to program. These models are summarized in Fig. 3.4.

3.2 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.

3.2.1 User Interfaces

A major task of most clients is to interact with a human user and a remote server. Supporting the interface to the user is a key feature of most clients. In many cases, the interface between a user and a remote server is relatively simple and integrated with the client hardware. For example, cellular phones have a simple display combined with a traditional set of keys for dialing numbers. More sophisticated ones that also offer e-mail facilities, may be equipped with a complete keyboard, an electronic pad, or a unit for speech recognition.

An important class is formed by graphical user interfaces. In the following pages, we first take a closer look at the X window system as an example of a more traditional graphical user interface. We then consider modern interfaces that support direct communication between applications.

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. 3.5. (In X terminology, the X kernel is referred to as the X server, whereas programs that make use of its features, are called X clients. To avoid confusion with standard client-server terminology, we refrain from using the notions of X server and X client.) X distinguishes between two types of application programs: normal applications and window managers. Normal applications generally request (through Xlib) the creation of a window on the screen, which they subsequently use for input and output processing. In addition, X will ensure that whenever an application's window is active, that is, when the mouse is pointing inside that window, that all events from the keyboard and the mouse are passed to the application.
A window manager is an application that is given special permission to manipulate the entire screen. Normal applications have to obey the restrictions on screen manipulation as implemented by the window manager. For example, the window manager may decide that windows may never overlap, or that windows should always be displayed in the same color. Consequently, the window manager determines the “look and feel” of the window system as a whole.

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 a network-oriented communication protocol by which an instance of Xlib can exchange data and events with the X kernel. This leads to very different types of client-server organizations by which the level of sophistication of the X terminals. In its most sophisticated form, the client machine contains numerous applications, including a window manager, and hardly any network communication is necessary.

It is important to realize that user-interface systems such as X essentially provide no more than a graphical interface to applications. The only information that basic user applications can expect from such systems are events identifying basic user actions that are directly related to the devices attached to the terminal. Examples of such events are those regarding keystrokes, mouse position, button operations, etc.

### Compound Documents

As we also mentioned in Chap. 1, modern user interfaces do a lot more than just window systems such as X. In particular, they 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.

### 3.2.2 Client-Side Software for Distribution Transparency

As we also mentioned in Sec. 1.5, 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 location from the user, and also transparently bind 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 remote objects. Replication of a remote object can be achieved by forwarding an invocation request to each replica, as shown in Fig. 3-6. The client’s proxy can transparently collect all responses and pass a single return value to the client application.

![Figure 3-6. A possible approach to transparent replication of a remote object using a client-side solution.](image)

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.

3.3 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 object servers. Object servers are important because they form the building block for implementing distributed objects.

3.3.1 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 endpoint, also called a port, at the machine where the server is running. Each server listens to a specific endpoint. How do clients know the endpoint of a service? One approach is to globally assign endpoints 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 endpoints have been assigned by the Internet Assigned Numbers Authority (IANA), and are documented in (Reynolds and Postel, 1994). With assigned endpoints, 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 endpoint. For example, a time-of-day server may use an endpoint that is dynamically assigned to it by its local operating system. In that case, a client will first have to look up the endpoint. One solution, as we saw in DCE, is to have a special daemon running on each machine that runs servers. The daemon keeps track of the current endpoint of each service implemented by a colocated server. The daemon itself listens to a well-known endpoint. A client will first contact the daemon, request the endpoint, and then contact the specific server, as shown in Fig. 3-7(a).

It is common to associate an endpoint with a specific service. However, actually implementing each service by means of a separate server may be a waste of
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 endpoint to which the client sends out-of-band data, while at the same time listening (with a lower priority) to the endpoint 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, 1996). 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.

In contrast, a stateful server does maintain information on its clients. 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. 7, 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.

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. 10, 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
3.3.2 Object Servers

After having taken a look at some general design issues, we now consider a special kind of server that is becoming increasingly important. An object server is a server tailored to support distributed objects. The important difference between a general object server and other (more traditional) servers, is that an object server by itself does not really provide a specific service. Specific services are implemented by the objects that reside in the server. Essentially, the server is implemented by the objects that reside in it. The server provides only the means to invoke local objects, based on requests from remote clients. As a consequence, it is relatively easy to change services by simply adding and removing objects. An object server thus acts as a place where objects live. An object consists of two parts: data representing its state and code forming the implementation of its methods. Whether or not these parts are separated, or whether method implementations are shared by multiple objects, depends on the object server. Also, there are differences in the way an object server invokes its objects. For example, in a multithreaded server, each object may be assigned a separate thread, or a separate thread may be used for each invocation request. These and other issues are discussed next.

Alternatives for Invoking Objects

For an object to be invoked, the object server needs to know which code to execute, on which data it should operate, whether it should start a separate thread to take care of the invocation, and so on. A simple approach is to assume that all objects look alike and that there is only one way to invoke an object. In essence, this is what DCE does. Unfortunately, such an approach is generally inflexible and often unnecessarily constrains developers of distributed objects.

A much better approach is for a server to support different policies. Consider, for example, transient objects. Recall that a transient object is an object that exists only as long as its server exists, but possibly for a shorter time. An in-memory, read-only copy of a file could typically be implemented as a transient object. Likewise, a calculator (possibly running on a high-performance server), could also be implemented as a transient object. A reasonable policy is to create a transient object at the first invocation request, and to destroy it as soon as no clients are bound to it anymore. The advantage of this approach is that a transient object will need a server’s resources only as long as the object is really needed. The drawback is that an invocation may take some time to complete, because the object needs to be created first. Therefore, an alternative policy is sometimes to create all transient objects at the time the server is initialized, at the cost of consuming resources even when no client is making use of the object.

In a similar fashion, a server could follow the policy that each of its objects is placed in a memory segment of its own. In other words, objects share neither code nor data. Such a policy may be necessary when an object implementation does not separate code and data, or when objects need to be separated for security reasons. In the latter case, the server will need to provide special measures, or require support from the underlying operating system, to ensure that segment boundaries are not violated. The alternative approach is to let objects at least share their code. For example, a database containing objects that belong to the same class can be efficiently implemented by loading the class implementation only once into the server. When a request for an object invocation comes in, the server need only fetch that object’s state from the database and execute the requested method.

Likewise, there are many different policies with respect to threading. The simplest approach is to implement the server with only a single thread of control. Alternatively, the server may have several threads, one for each of its objects. Whenever an invocation request comes in for an object, the server passes the request to the thread responsible for that object. If the thread is currently busy, the request is temporarily queued. The advantage of this approach is that objects are automatically protected against concurrent access: all invocations are serialized through the single thread associated with the object. Of course, it is also possible to use a separate thread for each invocation request, requiring that objects should have already been protected against concurrent access. Independent of using a thread per object or thread per method is the choice of whether threads are created on demand or the server maintains a pool of threads. Generally there is no single best policy.

Object Adapter

Decisions on how to invoke an object are commonly referred to as activation policies, to emphasize that in many cases the object itself must first be brought into the server’s address space (i.e., activated) before it can actually be invoked.
What is needed then, is a mechanism to group objects per policy. Such a mechanism is sometimes called an object adapter, or object wrapper, but is often just hidden away in a set of tools for building object servers. We adopt the term object adapter. An object adapter can best be thought of as software implementing a specific activation policy. The main issue, however, is that object adapters come as generic components to assist developers of distributed objects, and which need only to be configured for a specific policy.

An object adapter has one or more objects under its control. Because a server should be capable of simultaneously supporting objects that require different activation policies, several object adapters may reside in the same server at the same time. When an invocation request is delivered to the server, that request is first dispatched to the appropriate object adapter, as shown in Fig. 3-8.

![Diagram showing object server with three objects and server machine](image)

**Figure 3-8.** Organization of an object server supporting different activation policies.

An important observation is that object adapters are unaware of the specific interfaces of the objects they control. Otherwise, they could never be generic. The only issue that is important to an object adapter is that it can extract an object reference from an invocation request, and subsequently dispatch the request to the referenced object, but now following a specific activation policy. As is also shown in Fig. 3-8, rather than passing the request directly to the object, an adapter hands an invocation request to the server-side stub of that object. The stub, also called a skeleton, is normally generated from the interface definitions of the object, unmarshals the request and invokes the appropriate method.

As an example, consider an object adapter that manages a number of objects. The adapter implements the policy that it has a single thread of control for each of

SEC. 3.3

its objects. To interact with the object-specific skeletons that marshal and unmarshal requests, it expects that each skeleton implements the operation

```c
invoke(unsigned int in_size, char in_args[], unsigned* out_size, char* out_args[])
```

in which `in_args` is an array of bytes that needs to be unmarshaled by the stub. The array contains an identification of the method, along with values for all its parameters. The exact format of the array is known only to the stub, which is also responsible for the actual invocation. The parameter `in_size` specifies the length of `in_args`. In a similar fashion, all output is marshaled by the stub into an array `out_args` which is dynamically created by the stub. The length of the array is specified by the output parameter `out_size`. (Note that `invoke` is similar to the version discussed in the previous chapter, which was used for dynamic invocation.)

```c
/* Definitions needed by caller of adapter and adapter */
#define TRUE 1
#define MAX_DATA 65536

/* Definition of general message format. */
struct message {
    long source; /* sender's identity */
    long object_id; /* identifier for the requested object */
    long method_id; /* identifier for the requested method */
    unsigned size; /* total bytes in list of parameters */
    char* data; /* parameters as sequence of bytes */
};

/* General definition of operation to be called at skeleton of object */
typedef void (*METHOD_CALL)(unsigned char*, unsigned*, char*);

long register_object(METHOD_CALL call); /* register an object */
void unregister_object(long object_id); /* unregister an object */
void invoke_adapter(message *request); /* call the adapter */

Figure 3-9. The header.h file used by the adapter and any program that calls an adapter.

Fig. 3-9 shows the header file of the adapter. The most important part is the definition of the messages the adapter exchanges with remote clients. Each client is expected to marshal an invocation request into a message having five fields. Likewise, the adapter will return a response in a message having the same structure. The field `source` identifies the sender of the message. The `object_id` and `method_id` fields uniquely identify the object and the method, respectively, which are to be invoked. The input data that are to be passed to the stub, are contained in the array `data` of which the exact size is given by the field `size`. The results of the invocation are similarly later put into the `data` field of a new message.
The header file also contains the definition of what the adapter expects it can call at the server-side stub of an object by means of the `METHOD_CALL` type definition.

Finally, the adapter provides two procedures that can be called by a server to register and unregister objects at the adapter. Registration takes place by passing a pointer to the object-specific implementation of the `invoke` procedure, as implemented in the object’s stub. Registration returns a number that can be effectively used as an object identifier relative to the adapter. To unregister an object, the server merely passes this number when calling `unregister_object`. The actual call to the adapter is done through the procedure `invoke_adapter`, which requires an identifier for the object and an invocation request. The results will later be put into a separate buffer, as we explain next.

```c
typedef struct thread THREAD; /* Hidden definition of a thread. */

THREAD *create_thread(void (*body)(long tid), long thread_id);
/* Create a thread by giving a pointer to a function that defines the actual */
/* behavior of the thread, along with an integer that is used to */
/* uniquely identify the thread. */

void get_msg(unsigned *size, char **data);
void put_msg(THREAD *receiver, unsigned size, char *data);
/* Calling get_msg blocks the thread until a message has been put into its */
/* associated buffer. Putting a message in a thread's buffer is a nonblocking */
/* operation. */
```

Figure 3-10. The `thread.h` file used by the adapter for using threads.

To implement the adapter, we assume there is a thread package available that provides the necessary facilities for creating (and deleting) threads and for letting threads communicate. Communication between threads takes place by means of buffers. In particular, each thread has its own associated buffer from which it can remove a message by means of the blocking operation `get_msg`. Messages are appended to a buffer through the nonblocking operation `put_msg`. The main part of the header file of the thread package is shown in Fig. 3-10.

We now come to the actual implementation of the adapter, which is shown in Fig. 3-11. Each object has its own associated thread specified by the procedure `thread_per_object`. A thread starts by blocking until an invocation request is been put into its associated buffer. The request is immediately passed to the object’s stub by calling `invoke(object_id)` with the appropriate parameter values. The results of the object invocation are returned in the variable `results`, and will then have to be copied to a response message. That response message is constructed by first setting the fields `object_id` and `method_id`, and subsequently copying the results to the data field of the message. At that point, the response

```c
#include <header.h>
#include <thread.h>
define MAX_OBJECTS 100
#define NULL 0
#define ANY -1

METHOD_CALL invoke[MAX_OBJECTS]; /* array of pointers to stubs */
THREAD *root; /* demultiplexer thread */
THREAD *thread[MAX_OBJECTS]; /* one thread per object */

void thread_per_object(long object_id) {
    message req, res; /* request/response message */
    unsigned size; /* size of messages */
    char results; /* array with all results */

    while(TRUE) {
        get_msg(&size, (char*) &req); /* block for invocation request */

        /* Pass request to the appropriate stub. The stub is assumed to */
        /* allocate memory for storing the results. */
        (invoke(object_id))(req->size, req->data, &size, &results);

        res = malloc(sizeof(message)+size); /* create response message */
        res->object_id = object_id; /* identify object */
        res->method_id = req.method_id; /* identify method */
        res->size = size; /* set size of invocation results */
        memcpy(res->data, results, size); /* copy results into response */
        put_msg(root, sizeof(res), res); /* append response to buffer */
        free(req); /* free memory of request */
        free(results); /* free memory of results */
    }
}

void invoke_adapter(long oid, message *request) {
    put_msg(thread[oid], sizeof(request), request);
}
```

Figure 3-11. The main part of an adapter that implements a thread-per-object policy.

be handed over to the demultiplexer, as shown in Fig. 3-8. In our example, the demultiplexer is implemented by a separate thread referred to by the variable `root`. The implementation of `invoke_adapter` is now simple. The calling thread (i.e., the demultiplexer in our example) appends its invocation request to the buffer of the thread associated with the object that is required to be invoked. Later, the
3.4 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 and discuss the D’Agents system for mobile agents at the end of this section. Note that security issues concerning code migration are deferred to Chap. 8.

3.4.1 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. 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.
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. 3-12. 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.

![Figure 3-12. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.](image)

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 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. 8, 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 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 its initial state. This is what happens, for example, with Java applets. 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 powerful than weak mobility, but also much harder to implement. An example of a system that supports strong mobility is D’Agents, which we discuss later in this section.

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 often simpler to implement 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. 8.

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 just mentioned.

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.

To understand the implications that code migration has on the resource segment, Faggetta et al. 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 URL to refer to a specific Web site or when it refers to an FTP server by means of that server’s Internet address. In the same line of reasoning, references to local communication endpoints 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 endpoint.

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. 3-14.

<table>
<thead>
<tr>
<th>Resource-to-machine binding</th>
<th>Unattached</th>
<th>Fastened</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>By identifier</strong></td>
<td>MV (or GR)</td>
<td>GR (or MV)</td>
<td>GR</td>
</tr>
<tr>
<td><strong>By type</strong></td>
<td>CP (or MV,GR)</td>
<td>GR (or CP)</td>
<td>GR</td>
</tr>
<tr>
<td><strong>By value</strong></td>
<td>RB (or MV,CP)</td>
<td>RB (or GR,CP)</td>
<td>GR</td>
</tr>
</tbody>
</table>

**Figure 3-14.** Actions to be taken with respect to the references to local resources when migrating code to another machine.

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 establish a global reference.

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 computer server. Establishing a global reference to the multimedia workstation means setting up a communication path between the computer 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 easy is when migrating a process that is making use of a local communication endpoint. 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 endpoint 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 distributed shared memory mechanisms as discussed in Chap. 1. Obviously, this is not a really viable 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 rebinding 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.

### 3.4.3 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 its 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, perhaps after recompiling the original source. Also, we need to ensure that the execution segment can be properly represented at each platform.

Problems can be somewhat alleviated when dealing only with weak mobility. In that case, there is basically no runtime information that needs to be transferred between machines, so that it suffices to compile the source code, but generate different code segments, one for each potential target platform.
In the case of strong mobility, the major problem that needs to be solved is the transfer of the execution segment. The problem is that this segment is highly dependent on the platform on which the process is being executed. In fact, only when the target machine has the same architecture and is running exactly the same operating system, is it possible to migrate the execution segment without having to alter it.

The execution segment contains data that is private to the process, its current stack, and the program counter. The stack will partly consist of temporary data, such as values of local variables, but may also contain platform-dependent information such as register values. The important observation is that if we can avoid having execution depend on platform-specific data, it would be much easier to transfer the segment to a different machine, and resume execution there.

A solution that works for procedural languages such as C and Java is shown in Fig. 3-15 and works as follows. Code migration is restricted to specific points in execution of a program. In particular, migration can take place only when a function is called. A subroutine is a function in C, a method in Java, and so on. The runtime system maintains its own copy of the program stack, but in a machine-independent way. We refer to this copy as the migration stack. The migration stack is updated when a subroutine is called, or when execution returns from a subroutine.

When a subroutine is called, the runtime system marshals the data that have been pushed onto the stack since the last call. These data represent values of local variables, along with parameter values for the newly called subroutine. The marshaled data are then pushed onto the migration stack, along with an identifier for the called subroutine. In addition, the address where execution should continue when the caller returns from the subroutine is pushed in the form of a jump label onto the migration stack as well.

If code migration takes place at the point where a subroutine is called, the runtime system first marshals all global-program-specific data forming part of the current execution segment. Machine-specific data are ignored, as well as the current stack. The marshaled data are transferred to the destination, along with the migration stack. In addition, the destination loads the appropriate code segment containing the binary. In the runtime system architecture and operating system. The marshaled data belonging to the execution segment are unmarshaled, and a new runtime stack is constructed by unmarshaling the migration stack. Execution can then be resumed by simply entering the subroutine that was called at the original site.

It is clear that this approach works only if the compiler generates code to update the migration stack whenever a subroutine is entered or exited. The compiler also generates labels in the caller’s code allowing a return from a subroutine to be implemented as a (machine-independent) jump. In addition, we also need to support by preprocessor to insert the necessary code to maintain the migration stack.

Figure 3-15. The principle of maintaining a migration stack to support migration of an execution segment in a heterogeneous environment.
The only serious drawback of the virtual-machine approach is that we are generally stuck with a specific language, and it is often not one that has been used before. For this reason, it is important that languages for mobility provide interfaces to existing languages.

### 3.4.4 Example: D’Agents

To illustrate code migration, let us now take a look at a middleware platform that supports various forms of code migration. D’Agents formerly called Agent Tcl, is a system that is built around the concept of an agent. An agent in D’Agents is a program that can migrate between machines in a heterogeneous system. Here, we concentrate only on the migration capabilities of D’Agents, and return to a more general discussion on software agents in the next section. Also, we ignore the security of the system and defer further discussion to Chap. 8. More information on D’Agents can be found in (Gray, 1996b; Kotz et al., 1997).

#### Overview of Code Migration in D’Agents

An agent in D’Agents is a program that can migrate between different machines. In principle, programs can be written in any language, as long as the target machine can execute the migrated code. In practice, this means that programs in D’Agents are written in an interpretable language, notably, the Tool Command Language, that is, Tcl (Ousterhout, 1994), Java, or Scheme (Rees and Clinger, 1986). Using only interpretable languages makes it much easier to support heterogeneous systems.

A program, or agent, is executed by a process running the interpreter for the language in which the program is written. Mobility is supported in three different ways: sender-initiated weak mobility, strong mobility by process migration, and strong mobility by process cloning.

Weak mobility is implemented by means of the agent::submit command. An identifier of the target machine is given as a parameter, as well as a script that is to be executed at that machine. A script is nothing but a sequence of instructions. The script is transferred to the target machine along with any procedure definitions and copies of variables that the target machine needs to execute the script. At the target machine, a process running the appropriate interpreter is subsequently started to execute the script. In terms of the alternatives for code migration mentioned in Fig. 3.13, D’Agents thus provides support for sender-initiated weak mobility, where the migrated code is executed in a separate process.

To give an example of weak mobility in D’Agents, Fig. 3.16 shows part of a simple Tcl agent that submits a script to a remote machine. In the agent, the procedure factorial takes a single parameter and recursively evaluates the expression $n!$ for the factorial of its parameter value. The variables number and machine are assumed to be properly initialized (e.g., by asking the user for values), after which the agent calls agent::submit. The script

```tcl
proc factorial {n} {
    if { [incr n] <= 1 } { return 1; }
    set n [* [factorial [expr $n - 1]]]
    return $n
}
```

is sent to the target machine referred to by the variable machine, along with the description of the procedure factorial and the initial value of the variable number. D’Agents automatically arranges that results are sent back to the agent. The call to agent::receive establishes that the submitting agent is blocked until the results of the calculation have been received.

```tcl
    set number ... # tells which factorial to compute
    set machine ... # identify the target machine
    agent::submit $machine -proc factorial -vars number -script { factorial $number }
    agent::receive ... # receive the results (left unspecified for simplicity)
```

**Figure 3.16.** A simple example of a Tcl agent in D’Agents submitting a script to a remote machine (adapted from Gray, 1995).

Sender-initiated strong mobility is also supported, both in the form of process migration and process cloning. To migrate a running agent, the agent calls agent::jump specifying the target machine to which it should migrate. When agent::jump is called, execution of the agent on the source machine is suspended and its resource segment, code segment, and execution segment are marshaled into a message that is subsequently sent to the target machine. Upon arrival of that message, a new process running the appropriate interpreter is started. That process unmarshals the message and continues at the instruction following the previous call to agent::jump. The process that was running the agent at the source machine, exits.

An example of agent migration is given in Fig. 3.17, which shows a simplified version of an agent that finds out which users are currently logged in by executing the UNIX command who on each host. The behavior of the agent is given by the procedure all::users. It maintains a list of users that is initially empty. The list of hosts that it should visit is given by the parameter machines. The agent jumps to each host, puts the results of executing who in the variable users, and appends that to its list. In the main program, the agent is created on the current
proc all_users machines {
    set list ""
    foreach m $machines {
        agent_jump $m
        set users [exec who]
        append list $users
    }
    return $list
}

set machines ...
set this_machine ...
# Create a migrating agent by submitting the script to this machine, from where
# it will jump to all the others in $machines.
agent_submit this_machine -procs all_users --vars machines \
    -script { all_users $machines }
agent_receive ...

Figure 3-17. An example of a Tcl agent in D'Agents migrating to different machines where it executes the UNIX who command (adapted from Gray, 1995).

machine by submission, that is, using the previously discussed mechanisms for weak mobility. In this case, agent_submit is requested to execute the script

    all_users $machines

and is given the procedure and set of hosts as additional parameters.

Finally, process cloning is supported by means of the agent_fork command. This command behaves almost the same as agent_jump, except that the process running the agent at the source machine simply continues with the instruction following its call to agent_fork. Like the fork operation in UNIX, agent_fork returns a value by which the caller can check whether it is the cloned version (corresponding to the “child” in UNIX), or the original caller (i.e., the “parent”).

Implementation Issues

To explain some of the internal implementation details, consider agents that have been written in Tcl. Internally, the D'Agents systems consists of five layers, as shown in Fig. 3-18. The lowest layer is comparable to Berkeley sockets in the sense that it implements a common interface to the communication facilities of the underlying network. In D'Agents, it is assumed that the underlying system provides facilities for handling TCP messages and e-mail.

<table>
<thead>
<tr>
<th>Code Migration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agents</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Tc/Tk Interpreter</td>
</tr>
<tr>
<td>Common agent RTS</td>
</tr>
<tr>
<td>Server</td>
</tr>
<tr>
<td>TCP/IP</td>
</tr>
</tbody>
</table>

Figure 3-18. The architecture of the D'Agents system.

The next layer consists of a server that runs at each machine where D'Agents agents are executing. The server is responsible for agent management, authentication, and management of communication between agents. For the latter, the server assigns a location-unique identifier to each agent. Using the network address of the server, each agent can then be referred to by an (address, local-id)-pair. This low-level name is used to set up communication between two agents.

The third layer is at the heart of the D'Agents system, and consists of a language-independent core that supports the basic model of agents. For example, this layer contains implementations to start and end an agent, implementations of the various migration operations, and facilities for interagent communication. Clearly, the core operates closely with the server, but, in contrast to the server, is not responsible for managing a collection of agents running on the same machine.

The fourth layer consists of interpreters, one for each language supported by D'Agents. Each interpreter consists of a component for language interpretation, a security module, an interface to the core layer, and a separate module to capture the state of a running agent. This last module is essential for supporting strong mobility, and is discussed in more detail below.

The highest-level layer consists of agents written in one of the supported languages. Each agent in D'Agents is executed by a separate process. For example, when an agent migrates to machine A, the server there forks a process which will execute the appropriate interpreter for the migrating agent. The new process is then handed the state of the migrating agent, after which it continues where the agent had previously left off. The server keeps track of the processes it created using a local pipe, so that it can pass incoming calls to the appropriate process.

The more difficult part in the implementation of D'Agents, is capturing the state of a running agent and shipping that state to another machine. In the case of Tcl, the state of an agent consists of the parts shown in Fig. 3-19. Essentially, there are four tables containing global definitions of variables and scripts, and two tables for keeping track of the execution status.

There is a table for storing global variables needed by the interpreter. For example, there may be an event handler telling the interpreter which procedure to
call when a message from a specific agent arrives. Such an (event, handler)-pair is stored in the interpreter table. Another table contains global system variables for storing error codes, error strings, result codes, result strings, etc. There is also a separate table containing all user-defined global program variables. Finally, a separate table contains the definitions of the procedures associated with an agent. These procedure definitions need to migrate along with the agent in order to allow interpretation at the target machine.

The more interesting parts related to agent migration are the two stacks by which an accurate account is kept of the actual execution status of an agent. Basically, an agent is considered as a sequence of Tcl commands, possibly embedded in constructions such as loops, case statements, and so on. In addition, commands may be grouped into procedures. As is normal for any interpreted language, an agent is executed command by command. First consider what happens when a basic Tcl command is executed, that is, a command that is not a call to a user-defined procedure. The interpreter parses the command and builds a record that is to be pushed onto what is called the **command stack**. Such a record contains all the necessary fields to actually execute the command, such as its parameter values, a pointer to a procedure implementing the command, and so on. This record is then pushed onto the stack, after which it can be handed over to the component responsible for actually executing the command. In other words, the command stack gives a precise account of the current execution status of an agent.

Tcl also supports user-defined procedures. In addition to the command stack, the runtime environment of D’Agents keeps track of a stack of activation records, also called call frames. A call frame in D’Agents contains a table of variables on which the procedure was called. A call frame is created only as the result of a procedure-call command as pushed onto the procedure call, and as such is related to a procedure-call command as pushed onto the command stack. The call frame keeps a reference to its associated command.

Now consider what happens, for example, when an agent calls agent_jump. When an agent migrates to another machine. At that point, the complete state of the agent as just described is marshaled into a series of bytes. In other words, all four tables and the two stacks are put together into a single array of bytes and shipped to the target machine. The D’Agents server on the target machine subsequently creates a new process running the Tcl interpreter. That process is handed the marshaled data, which it then unmarshals into the state the agent was in when it called agent_jump. By simply popping the command from the top of the command stack, execution continues exactly where it had left off.

### 3.5 SOFTWARE AGENTS

So far, we have been looking at processes from very different angles. First, we concentrated on one of the essential issues, namely the thread(s) of control within a process. From the perspective of communication, we took a closer look at the general organization of clients as well as servers. Finally, we considered mobility of programs and processes. These more or less independent views on processes come together in what are commonly referred to as software agents: autonomous units capable of performing a task in collaboration with other, possibly remote, agents.

Agents are playing an increasingly important role in distributed systems. However, very similar to the fact that there was only an intuitive notion of what exactly a process was (see, for example, Oranick, 1972), software agents have yet to be precisely defined. In this section, we take a closer look at what agents are and their role in distributed systems.

#### 3.5.1 Software Agents in Distributed Systems

There is much controversy concerning what exactly an agent is. Staying in line with the description given in (Jennings and Woolridge, 1998), we define a **software agent** as an autonomous process capable of reacting to, and initiating changes in, its environment, possibly in collaboration with users and other agents. The feature that makes an agent more than just a process is its capability to act on its own, and, in particular, to take initiative where appropriate.

Our definition of a software agent is rather broad, and many different types of processes can easily be called an agent. Instead of trying to come to a better definition, it makes more sense to look at different types of agents. Again, several attempts in the literature have been made to develop a taxonomy of software agents, but it seems hard for researchers to reach agreement on a single taxonomy. Besides being autonomous, an important aspect of agents is that they should also be able to cooperate with other agents. The combination of autonomy and cooperation leads to the class of collaborative agents (Nwana, 1996). A **collaborative agent** is an agent that forms part of a multiagent system, in which agents...
seek to achieve some common goal through collaboration. A typical application
where collaborative agents could be used is arranging a meeting. Each attendee is
represented by an agent that has access to that user's personal agenda. Given all
the individual constraints with respect to time, travel, place, and so on, the
separate agents would collaborate in setting up a meeting. From the perspective of
distributed systems development, exactly which information is exchanged, and
how that is processed is of less concern. Important is how communication takes
place. We return to interagent communication below.

Many researchers also separate mobile agents from other agent types. A
mobile agent is simply an agent having the capability to move between different
machines. In terms of the discussion on code migration in the previous section,
mobile agents often require support for strong mobility, although this is not
strictly necessary. The requirement for strong mobility comes from the fact that
agents are autonomous and actively interact with their environment. Moving an
to another machine can hardly be done without considering its execution
state. However, as demonstrated by the D'Agents system, the combination of
agents and weak mobility is also useful. Note that mobility is a feature of agents
in general and does not lead to an exclusive class of its own. For example, it
makes sense to talk about mobile collaborative agents. A good example of
practical use of mobile agents is given in (Brewington et al., 1999), in which the authors
describe how mobile agents are used to retrieve information distributed across a
large heterogeneous network such as the Internet.

The ability to collaborate with other agents or to move between different
machines is a system property of agents. They tell us nothing about what the
agent can do. When taking a look at an agent's functionality, other classes can be
distinguished as well.

A generally recognized class is formed by interface agents, which are agents
that assist an end user in the use of one or more applications. A generally accepted
distinguishing property of an interface agent is that it has learning capabilities
(Maes, 1994; Nwana, 1996). The more often it interacts with the user, the better
its assistance becomes. In the context of distributed systems, an example of
interesting interface agents are those that seek interaction with agents for users in
the same community. For example, special interface agents exist that actively
seek to bring buyers and sellers together. By getting an increasingly better under-
standing of what its owner is looking for, or has to offer, such an interface agent
should improve on selecting a proper group of peers.

Closely related to interface agents are information agents. The main func-
tion of these agents is to manage information from many different sources.
Managing information includes ordering, filtering, collating, and so on. What
makes information agents important in distributed systems is that they operate on
information from physically different sources. Stationary information agents typi-
ically operate on incoming data streams. For example, an e-mail agent may be
capable of filtering unwanted mail from its owner's mailbox, or automatically
distributing incoming mail into appropriate subject-specific mailboxes. In con-
trast, mobile information agents generally roam the network on behalf of their
owner to collect required information.

In summary, agents can often be characterized by a number of properties, as
shown in Fig. 3-20 (see also Franklin and Graesser, 1996). Further distinctions
between agents are made by taking a look at how they actually operate from the
perspective of artificial intelligence. For a brief overview, see (Hayes, 1999;

<table>
<thead>
<tr>
<th>Property</th>
<th>Common to all agents?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autonomous</td>
<td>Yes</td>
<td>Can act on its own</td>
</tr>
<tr>
<td>Reactive</td>
<td>Yes</td>
<td>Responds timely to changes in its environment</td>
</tr>
<tr>
<td>Proactive</td>
<td>Yes</td>
<td>Initiates actions that affect its environment</td>
</tr>
<tr>
<td>Communicative</td>
<td>Yes</td>
<td>Can exchange information with users and other agents</td>
</tr>
<tr>
<td>Continuous</td>
<td>No</td>
<td>Has a relatively long life span</td>
</tr>
<tr>
<td>Mobile</td>
<td>No</td>
<td>Can migrate from one site to another</td>
</tr>
<tr>
<td>Adaptive</td>
<td>No</td>
<td>Capable of learning</td>
</tr>
</tbody>
</table>

Figure 3-20. Some important properties by which different types of agents can
be distinguished.

3.5.2 Agent Technology

Having only the notion of what agents are is not really helpful if there is no
support available for actually developing agent systems. An important issue is
then if we can isolate generally-used components of agents in distributed systems,
and incorporate these components into, for example, middleware. As a starting
point, the Foundation for Intelligent Physical Agents (FIPA) is developing a
general model for software agents. In this model, agents are registered at, and
operate under the regime of an agent platform as shown in Fig. 3-21. An agent
platform provides the basic services needed for any multiagent system. These
facilities include those for creating and deleting agents, facilities to locate agents,
and facilities for interagent communication.

An agent management component keeps track of the agents for the associated
platform. It provides the facilities for creating and deleting agents, but also for
looking up the current endpoint for a specific agent. In this sense, it provides a
naming service by which a globally unique identifier is mapped to a local com-
munication endpoint. Name services are discussed in detail in the next chapter.

There is also a separate local directory service by which agents can look up
what other agents on the platform have to offer. The directory service in the FIPA
model is based on the use of attributes. What this means is that an agent provides
Likewise, a message can have the purpose to respond to a previously sent request message. As another example, some messages can be sent to inform the recipient of an event, or to propose something in the act of negotiation. Several purposes of messages in an ACL developed by FIPA are listed in Fig. 3-22.

<table>
<thead>
<tr>
<th>Message purpose</th>
<th>Description</th>
<th>Message content</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFORM</td>
<td>Inform that a given proposition is true</td>
<td>Proposition</td>
</tr>
<tr>
<td>QUERY-IF</td>
<td>Query whether a given proposition is true</td>
<td>Proposition</td>
</tr>
<tr>
<td>QUERY-REF</td>
<td>Query for a given object</td>
<td>Expression</td>
</tr>
<tr>
<td>CFP</td>
<td>Ask for a proposal</td>
<td>Proposal specifics</td>
</tr>
<tr>
<td>PROPOSE</td>
<td>Provide a proposal</td>
<td>Proposal</td>
</tr>
<tr>
<td>ACCEPT-PROPOSAL</td>
<td>Tell that a given proposal is accepted</td>
<td>Proposal ID</td>
</tr>
<tr>
<td>REJECT-PROPOSAL</td>
<td>Tell that a given proposal is rejected</td>
<td>Proposal ID</td>
</tr>
<tr>
<td>REQUEST</td>
<td>Request that an action be performed</td>
<td>Action specification</td>
</tr>
<tr>
<td>SUBSCRIBE</td>
<td>Subscribe to an information source</td>
<td>Reference to source</td>
</tr>
</tbody>
</table>

Figure 3-22. Examples of different message types in the FIPA ACL (FIPA, 1998a), giving the purpose of a message, along with the description of the actual message content.

The essence of an ACL is, of course, that the sending and receiving agent both have at least the same understanding of the purpose of a message. Moreover, the purpose of a message often determines the reaction of the receiver. For example, when being asked for a proposal by means of a message having CFP in its header, the receiver is expected to actually respond with a proposal, that is a message with purpose PROPOSE. In this sense, an ACL actually defines a high-level communication protocol between a collection of agents.

Like most communication protocols, ACL messages consist of a header and the actual content. The header contains a field that identifies the purpose of the message, along with fields for identifying the sender and receiver. Also like many communication protocols, the message content is separated from, and independent of, the rest. In other words, the message content is assumed to be specific to the communicating agents. An ACL does not prescribe the format or language in which the message content is expressed.

What is necessary, then, is that enough information be provided to allow the receiving agent to properly interpret the content. To that end, an ACL message header can also contain a field to identify the language or encoding scheme for the content. This approach works fine as long as the sender and receiver have a common understanding how to interpret the data, or more precisely, the symbols in a message. When there is no such common understanding, an additional field is sometimes used to identify a standardized mapping of symbols to their meaning. Such a mapping is commonly referred to as an ontology.
To give a simple example, Fig. 3.23 shows a message expressed in FIPA ACL, used to inform an agent about Dutch royalty relationships. To identify the sending and receiving agent, each agent has a name that consists of several components. For example, max@http://fanclub-beatrix.royalty-spotters.nl:7239 may be used to refer to an agent called max residing on an agent platform with the DNS name fanclub-beatrix.royalty-spotters.nl. To communicate with the agent, the platform name will first have to be resolved by DNS to an IP address. Furthermore, the name specifies that communication should proceed by sending HTTP messages to a server on that host that is listening on port number 7239. In our example, agent max sends an informational message to an agent called elke, residing at a platform named royalty-watcher.uk. Messages should be sent using the IIOP protocol (which we discuss in Chap. 9), and sent to port number 5623.

The other fields in the message are related to its content. The language field specifies that the message content is expressed as a series of Prolog statements, whereas the ontology field identifies that those Prolog statements are to be semantically interpreted as genealogy information. Consequently, the receiving agent should now know that the statement

\[
\text{female(beatrix)}
\]

means that beatrix is the name of a woman, whereas

\[
\text{parent(beatrix, juliana, bernhard)}
\]

means that the mother of beatrix is named juliana, and that the father is named bernhard.

### 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.

Object servers form a special class. In essence, an object server is a process that has several objects placed in its address space, and for which it is willing to accept invocation requests. What makes an object server somewhat special is that there are many ways in which it can invoke objects. For example, a server can start a separate thread for each invocation request. Alternatively, it can use a thread per object, or even just a single thread for all its objects. Different invocation policies can be handled by the same server by making use of an object adapter. In essence, an object adapter is a component that implements exactly one invocation policy. There can be several object adapters per server.

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 perhaps the best solution to handle heterogeneity is to use virtual machines, by which heterogeneity is effectively hidden away through interpretative code.
A software agent, finally, is a special kind of process, which operates as an autonomous unit, but is capable of cooperation with other agents. From a distributed systems perspective, what separates agents from normal processes is their interaction by means of an application-level communication protocol called an agent communication language (ACL). In an ACL, a strict distinction is made between the purpose of a message and its content. An ACL defines a high-level communication protocol: a sent message generally prescribes a specific reaction from the receiver based only on the purpose of the message.

**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 file 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. Proxies can support replication transparency by invoking each replica, as explained in the text. Can (the server side of) an object be subject to a replicated invocation?

8. Constructing a concurrent server by spawning a process has some advantages and disadvantages compared to multithreaded servers. Mention a few.

9. Sketch the design of a multithreaded server that supports multiple protocols using sockets as its transport-level interface to the underlying operating system.

10. How can we prevent an application from circumventing a window manager and thus being able to completely mess up a screen?

11. Explain what an object adapter is.

12. Mention some design issues for an object adapter that is used to support persistent objects.

13. Change the procedure thread_main_object in the example of the object adapters, so that all objects under control of the adapter are handled by a single thread.

14. Is a server that maintains a TCP/IP connection to a client stateful or stateless?

15. 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?

16. To what extent does Java RMI rely on code migration?

17. 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.

18. In Fig. 3-13 it is suggested that strong mobility cannot be combined with executing migrated code in a target process. Give a counterexample.

19. Consider a process P that requires access to file F that 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?

20. Each agent in D’Agents is implemented by a separate process. Agents can communicate primarily through shared files and by means of message passing. Files cannot be transferred across machine boundaries. In terms of the mobility framework given in Sec. 3.4, which parts of an agent’s state, as given in Fig. 3-19, comprise the resource segment?

21. Compare the architecture of D’Agents with that of an agent platform in the FIPA model.

22. Where do agent communication languages (ACLs) fit into the OSI model?

23. Where does an agent communication language fit into the OSI model, when it is implemented on top of a system for handling e-mail, such as in D’Agents? What is the benefit of such an approach?

24. Why is it often necessary to specify the ontology in an ACL message?