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, followed by a discussion on how to deal with the local resources that a migrating program uses. A particularly hard problem is migrating code in heterogeneous systems, which is also discussed. To make matters concrete, we 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. Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well.

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

Consider, for example, a client-server system in which the server manages a huge database. If a client application needs to do many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. Otherwise, the network may be swamped with the transfer of data from the server to the client. In this case, code migration is based on the assumption that it generally makes sense to process data close to where those data reside.

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

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

Besides improving performance, there are other reasons for supporting code migration as well. The most important one is that of flexibility. The traditional approach to building distributed applications is to partition the application into different parts, and deciding in advance where each part should be executed. This approach, for example, has led to the different multitiered client-server applications discussed in Chap. 1.

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

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server. At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. This principle is shown in Fig. 3-7. This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine. Different solutions are discussed below and in later chapters.

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

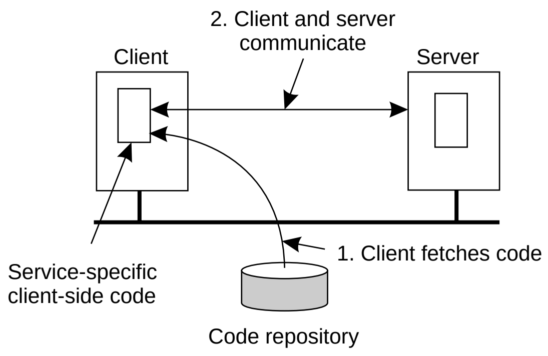
2. Client and server communicate
Client
Server
1. Client fetches code
Service-specific client-side code
Code repository

Figure 3-7. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

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 (Fugetta 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 and letting that child continue on a remote machine. The benefit of cloning is that the model closely resembles the one that is already used in many applications. The only difference is that the cloned process is executed on a different machine. In this sense, migration by cloning is a simple way to improve distribution transparency.

The various alternatives for code migration are summarized in Fig. 3-8.

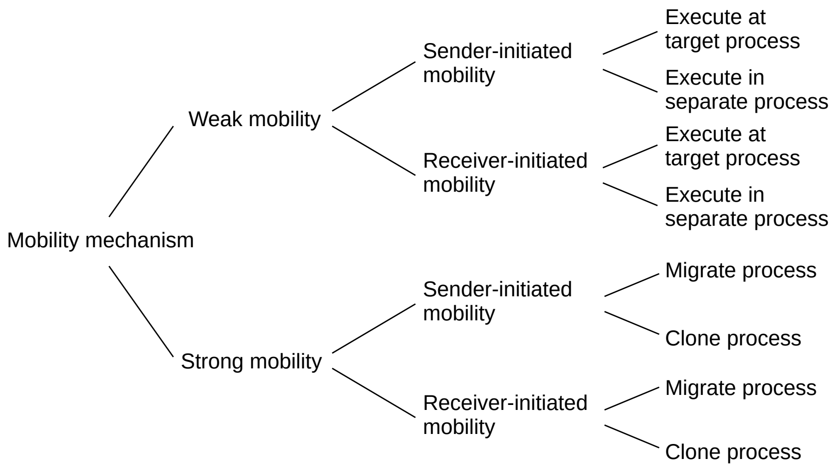
Mobility mechanism
Weak mobility
Strong mobility
Sender-initiated mobility
Receiver-initiated mobility
Sender-initiated mobility
Receiver-initiated mobility
Execute at target process
Execute in separate process
Execute at target process
Execute in separate process
Migrate process
Clone process
Migrate process
Clone process

Figure 3-8. Alternatives for code migration.

3.4.2 Migration and Local Resources

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

To understand the implications that code migration has on the resource segment, Fuggetta et al. 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. 3-9.

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

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

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

Figure 3-9. Actions to be taken with respect to the references to local resources when migrating code to another machine.

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

Another example of where establishing a global reference is not always that easy, is when migrating a process that is making use of a local communication 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 really a 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 rebind the process to a locally available resource of the same type. Only when such a resource is not available, will we need to copy or move the original one to the new destination, or establish a global reference.

3.4.3 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 their own operating system and machine architecture. Migration in such systems requires that each platform is supported, that is, that the code segment can be executed on each platform, 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. 3-10 and works as follows. Code migration is restricted to specific points in the execution of a program. In particular, migration can take place only when a next subroutine 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.

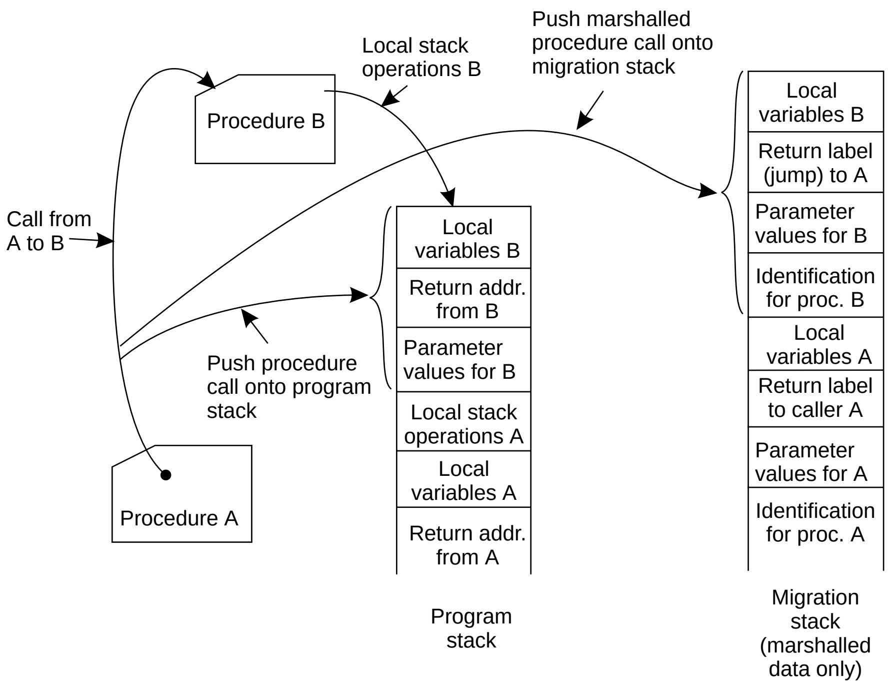

Figure 3-10. The principle of maintaining a migration stack to support migration of an execution segment in a heterogeneous environment.

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 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 binaries fit for its machine 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 a suitable runtime system. Nevertheless, there are a number of systems that have successfully exploited these techniques. For example, Dimitrov and Rego (1998) show how migration of C/C++ programs in heterogeneous systems can be supported by slightly modifying the language, and using only a preprocessor to insert the necessary code to maintain the migration stack.

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

About 20 years later, code migration in heterogeneous systems is being attacked by scripting languages and highly portable languages such as Java. All such solutions have in common that they rely on a virtual machine that either directly interprets source code (as in the case of scripting languages), or otherwise interprets intermediate code generated by a compiler (as in Java). Being in the right place at the right time is also important for language developers.

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. 3-8, 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. 3-11 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 that calculates 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

```
factorial $number
```

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.

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

```
proc factorial n {
    if { $n ≤ 1 } { return 1; }                    # fac(1) = 1
    expr $n * [ factorial [ expr $n – 1] ]         # fac(n) = n * fac(n–1)
}

set number ...        # tells which factorial to compute
set machine ...       # identify the target machine

agent_submit $machine –procs factorial –vars number –script { factorial $number }

agent_receive ...     # receive the results (left unspecified for simplicity)
```

Figure 3-11. A simple example of a Tcl agent in D'Agents submitting a script to a remote machine (adapted from Gray, 1995).

call to agent_jump. The process that was running the agent at the source machine, exits.

```
proc all_users machines {
    set list ""                     # Create an initially empty list
    foreach m $machines {           # Consider all hosts in the set of given machines
        agent_jump $m               # Jump to each host
        set users [exec who]        # Execute the who command
        append list $users          # Append the results to the list
    }
    return $list                    # Return the complete list when done
}
set machines ...                    # Initialize the set of machines to jump to
set this_machine ...                # Set to the host that starts the agent

# 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 ...                   # receive the results (left unspecified for simplicity)
```

Figure 3-12. An example of a Tcl agent in D'Agents migrating to different machines where it executes the UNIX *who* command (adapted from Gray, 1995).

An example of agent migration is given in Fig. 3-12, 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 set 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 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. 3-13. 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.

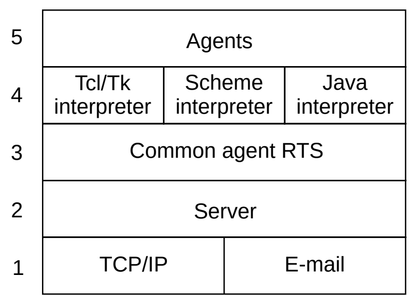

Figure 3-13. The architecture of the D'Agents system.

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 that 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. 3-14. Essentially, there are four tables containing global definitions of variables and scripts, and two stacks for keeping track of the execution status.

State	Description
Global interpreter variables	Variables needed by the interpreter of an agent
Global system variables	Return codes, error codes, error strings, etc.
Global program variables	User-defined global variables in a program
Procedure definitions	Definitions of scripts to be executed by an agent
Stack of commands	Stack of commands currently being executed
Stack of call frames	Stack of activation records, one for each running command

Figure 3-14. The parts comprising the state of an agent in D'Agents.

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 series 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 local to the procedure, along with the names and values of the parameters by which the procedure was called. A call frame is created only as the result of a 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, by which the 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.