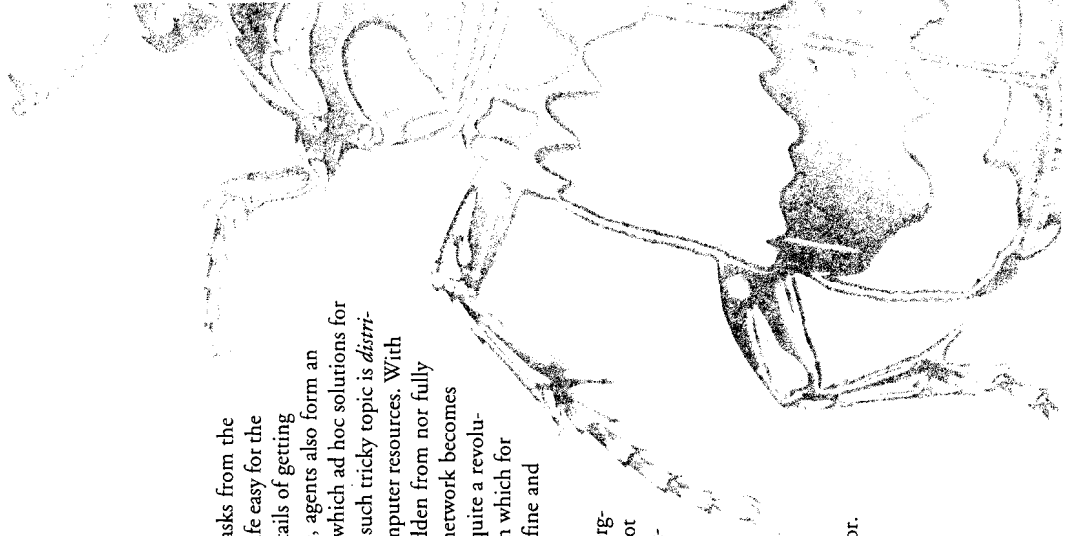


By *Giovanna Di Marzo, Murhimanya Mubugusa, and Christian F. Tschudin*

# Agent Mobility

The agent metaphor emphasizes delegating tasks from the human user to an *electronic agent*. This makes life easy for the user, as he need not care about the tedious details of getting some computer tasks done. On the other side, agents also form an abstraction layer for the programmer, behind which ad hoc solutions for tricky technical questions can be hidden. One such tricky topic is *distribution*—how to handle today's distributed computer resources. With mobile agents, distribution is neither fully hidden from nor fully exposed to the user. As a matter of fact, the network becomes programmable via the mobile agents. This is quite a revolution when compared to the classic approach, in which for each distributed application one needed to define and standardize a new protocol.

This chapter gives a brief overview of the emerging mobile code technology. Agent mobility is not mastered yet and currently no conclusive treatment of this topic can be presented, but in this chapter we will try to sketch the huge design space created by the mobile agent concept. It is important to understand the technological choices being made today, because they will help you choose among the future mobile agent systems, each having a different flavor.



## Why Mobility?

Mobile agents are often mentioned in conjunction with mobile computing equipment like handheld personal communicators or personal digital assistants. Such devices' defining characteristics are not only their limited computing and memory capacity, but also the fact that most of the time they are used offline. This means that these devices have only sporadic network connectivity, ruling out the use of important Internet applications that require session-long connections. Here, mobile agents provide a nice solution: The user delegates information-seeking tasks to an agent, which the user connects to the network. The agent jumps into the network and starts its task. The next time the user connects, the agent comes back and delivers the result. Mobile agents thus form a variant of mobile computing that contrasts with ordinary approaches like the extension of the Internet to mobile equipment through special protocols, cellular phones, and so on.

Mobility is also an important field where intelligent agents can be applied, because mobile agents must deal with a continually fluctuating network in which they must look for services at unknown locations and with vaguely known interfaces. Adaptability will become a prime attribute for mobile agents. Furthermore, because the agent metaphor does not fully hide the physical distribution aspect, agent mobility is the key to the agent's *distribution awareness*. Only by taking into account (instead of hiding) distribution, and by giving to the agent control over its physical placement, can agents optimize their network resource usage.

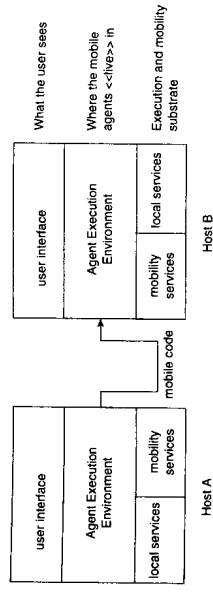
Finally, the mobile agent metaphor has the potential to become the basis of *all* distributed computing activities. Currently, designers are mainly creating mobile agents for the applications level, but the underlying technology can also be used at the (distributed) operating systems and network services levels in general. However, there must be much research done before such a unified architecture can become possible.

## The Foundation of Agents: Mobile Code

Making agents mobile implies that there must be some *transport infrastructure* that physically moves the agent's bits and bytes from one location to the other. Agents are active entities that do some remote jobs. Therefore, agents are mobile threads of execution that can spawn through the network, and support for code to move is needed. The agent may be intelligent or not; it may be an interface assistant. As soon as it is mobile, it relies on code mobility.

Looking at the general architecture of mobile code systems (see Figure 20.1), we can distinguish three different spheres. At the top level, agents communicate with their (human) owners to present results or to ask for more precise instructions. This is what the user will see. The agents themselves *live* in an intermediate layer composed of all the computers that provide for them an execution environment. Finally, these environments are linked through common mobility services that enable agents to hop from node to node, accessing the remote host's local services.

**Figure 20.1.**  
General software architecture  
of a mobile agent system.



Mobility for agents is a fine concept for the user, but how do we achieve it? What is the price of it? Who is in charge of doing distributed computing "the mobile way"? As we will show in this chapter, opinions differ greatly on these questions. One theme of this debate is the role of the subsystem that implements the execution and mobility substrate.

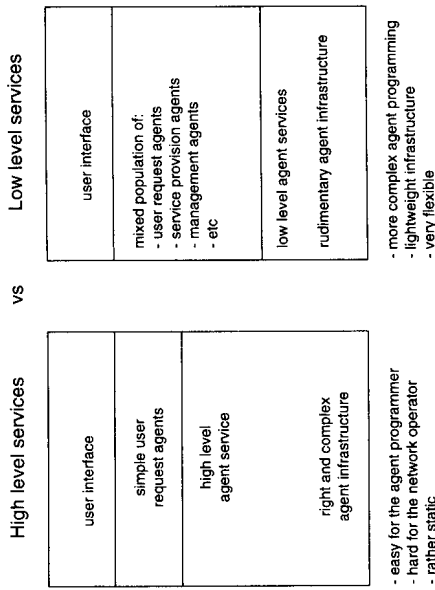
Conceptually it seems so easy: You just build a special module that handles mobility. However, with this approach you will end up with a huge and clumsy system, because each user community (and thus agent population) has different requirements, making the one-for-all subsystem difficult to design and operate. The other approach is to provide only minimal mobility support, making life easy for the network operators. However, the mobile agent programmer will have the burden of implementing or assembling the services that he requires.

The first model, shown in Figure 20.2, is currently the most prominent, for several reasons. First, it gives more control to the infrastructure provider—this is an essential asset for security matters, for example. Second, it makes it easy to show working agents, because inside an agent you mainly assemble high-level agent services, leaving the problem of implementing them to state-of-the-art software technology. The drawback, however, is the resulting rigid system: You can offer advanced services only by modifying virtually *all* hosts running the agent infrastructure software.

On the other side, we have the minimalist approach of putting into the agent infrastructure only rudimentary services, with the hope that these basic services will be sufficient for a long time. Advanced and new services are then implemented by adding service provision agents to the system *without* changing the agent infrastructure. This evolving environment even allows for economic competition between different service agent populations. Although this second approach is far-sighted, it has the drawback that many new techniques of agent-based service provision have yet to be developed.

Choosing the right level of agent services is a strategic and perhaps even a philosophical question—usually, debates and vendors' arguments concentrate more on technical issues of mobile code. Equipped with this general background, we now look at such tangible issues in more detail, showing some of the important design dimensions for mobile agent systems.

**Figure 20.2.**  
The debate on the right level of mobility support.



## Issues for Mobile Code Support

We mentioned that the mobile code concept is revolutionizing our way of seeing and doing distributed computing; computer science research has just started to explore this concept. This also means that we are faced with many new problems introduced by mobile code that were unknown until now in distributed computing. Security, for instance, looks quite different from an agent's point of view. The following sections establish a list of topics that every mobile code system designer must consider. Note that many of these items are heavily interdependent and sometimes contradictory: The art of designing good mobile code systems lies in the capacity to form optimal bundles.

### Design Dimension 1: Instruction Set and Agent Language

Shipping mobile code through the network makes sense only if the code can actually be executed on the remote side. Sending machine code is not a good choice, as only machines at which the code was targeted could understand it. That is why mobile agents are specified in a CPU-independent form. A standard approach is to define a fictitious computer (an instruction set) and to put an interpreter in every host's execution and mobility substrate. Agents are then specified in this intermediate instruction set.

The instruction set need not to be as low-level as for a real CPU. There are many agent systems that use high-level programming languages as their instruction set. This allows for a very comfortable programming environment, similar to popular software structuring styles like object-orientation. It is also possible to take existing scripting languages like Tcl and to reuse them in the agent domain.

Some systems use two levels of languages. At a low-level there are simple instructions based on a fictitious CPU model, and in parallel there is an object-oriented high-level language actually used by programmers. Most of the time, only the low-level instructions are sent through the network—special compiler programs translate the programmer's code into the low-level bytecodes.

More on this topic can be found in the section "Mobility Support in Agent Languages and Environments."

### Design Dimension 2: Host Security

Mobile agents very much resemble software viruses or network worms that have become notorious for their devastating effects and the difficulty in mastering them. Suspicion is justified, because the mobile code concept relies on remote hosts executing potentially alien code. Each host, thus, wants to protect itself against possible malicious attacks through mobile agents.

A first measure is to build a trust infrastructure and to enforce strict control on who may inject mobile agents into the system. This means that each mobile agent is authenticated by the system on arrival by checking its provenance and requesting third parties to certify its harmlessness. Note that this is mainly an organizational measure enforced by technical means, and that in case of administrative errors it cannot exclude worm-like scenarios. This is why some approaches choose not to build too much security into the execution and mobility support, and instead rely on other design choices (such as resource control).

Fortunately for the host's security, agents usually are interpreted (as they are specified in CPU-independent form), excluding, in principle at least, modifications to the interpreter software itself. However, if the instruction set includes special instructions that make the host's internal resources available to agents (which may be a very convenient feature) or that let agents execute arbitrary (machine code) programs, they require strict access controls based on passwords or other authentication techniques.

### Design Dimension 3: Agent Security

Protecting only the hosts is not sufficient security, because the mobile agents can be the target of malicious attacks too. The most difficult part is the agent's privacy, which is essential when agents carry electronic cash, copyrighted algorithms, or other secrets (such as passwords) with them. In fact, agents must be protected against malicious hosts and interceptors trying to rob them. That is why authentication is usually two-way, enabling only a successfully authenticated host to execute an agent (with other hosts failing to decrypt it). However, this is an organizational measure that cannot prevent accidents in the trust infrastructure.

Consider, for example, an agent that is responsible for buying a given service for its owner if the service costs less than \$100. The agent visits an execution environment offering the service

at \$90, but the environment looks at the agent code and determines the agent's intention. The environment then raises the service's price to the maximum value acceptable by the agent, and charges the agent \$99 instead of \$90.

Another problem for agents is sharing a host's resources with other agents. Here, agents would like some safety guarantees for successfully completing their task. Other agents may try to prevent this by modifying shared code or by hogging scarce resources.

### Design Dimension 4: Name Space Conventions

Bringing together hosts that all understand a common agent instruction set is not sufficient. Moreover, we need conventions about a common naming scheme for special computing resources. Although each host may differ in what it offers to agents, there must be a uniform way to discover what is present and how to access it, so that the agent can do its job. In an object-oriented environment, these are mainly code libraries installed in each host. Without them, no agent can run. However, resource names are also required for document databases, special agent services, and inter-agent communication.

Inter-agent communication is essential for implementing truly distributed agent-based applications. Otherwise, we would be restricted to one-task/one-agent applications, a restriction that would severely limit the mobile code concept's potential. What we need are not only means but also conventions on how agents can meet and exchange data. Some platforms offer rich concepts like "places" where two physically present agents can interact. Other environments provide simple shared memory, letting agents deposit and fetch data in an asynchronous way.

### Design Dimension 5: Resource Control

Resource control is required to operate smoothly an inherently concurrent mobile agent system. Resource control is a complex and difficult task, especially when it must take into account time constraints. For example, an agent can request to be scheduled in such a way that it completes its execution within a fixed time-frame. Different agents will have different requirements, and satisfying all of them can become a real challenge.

In the first place, the host's limited and shared resources like memory, CPU time, and network bandwidth must be assigned to the mobile agents in a fair way. Second, resource control is essential for charging the agent's owner for resource consumption. Third, resource control, or more specifically *consumption limits*, are in the agent's user's best interests, protecting him from unexpected bills.

Again, different solutions have been proposed in the various areas. Simple techniques like a decreasing hop counter built into each agent can prevent agents from looping in the network without making progress. CPU time limits inside a host can prevent programming errors from draining precious computing power. Electronic currencies, also used for paying local services (such as databases), can be used to enable agents to buy special resources on demand, helping

to better match resource requests and availability. Resource control is in its infancy, but will become, like security, a central element of mobile agent systems.

### Design Dimension 6: Programming Support and Agent Steering

Like any distributed system, agent-based applications need monitoring, tracing, and debugging. Once the user unleashes an agent, he has virtually no means of steering its future execution. This is a difficult situation for the agent developer, who essentially injects the agent into a distributed blackbox.

Having more control over a formerly unleashed agent may also be in the user's best interest. For example, a user might want to cancel a very time-consuming (and costly) search task. He needs some way to kill the search agent. Again the question is at what level this electronic leash should be implemented. Some systems offer explicit aids, whereas others fully rely on the programmer's ability to build the required functionality into the agent. Support for remote debugging, compilation on the fly, and code optimization are other areas where agent systems differ.

### Design Dimension 7: Efficiency

In the design dimensions discussed earlier, concerns about efficiency heavily influenced an agent system's design. Code mobility is expensive in terms of computer power: Freezing an agent's state, packaging and transferring it, authenticating the incoming agent, setting up the agent interpreter, and interpreting each instruction constitute formidable overhead. Each design decision must, therefore, be evaluated in order to minimize these costs.

This means that agent use may for a long time be restricted to very specific application niches where client/server models are not appropriate or are too complex. Working toward efficient agent systems is also a vital task for the system providers; otherwise, scaling is not possible.

Demonstrating an efficient system with thousands of users and hundreds of concurrent agents is not a problem. The problem occurs when this system is scaled at a worldwide base with millions of agents crawling through the networks. Agent system designers are in an uncertain situation similar to the one that early Internet engineers experienced in the 1970s. Only the reality test will show whether the basic design decisions were right.

### The Interworking of Heterogeneous Mobile Agents

Besides the programming language that expresses the agent behavior, most agents will have an inference system that they will use to reason and to enrich their knowledge. Two agents are considered heterogeneous if their behavior is expressed in different programming languages, if their knowledge is represented in different languages, or if they do not have the same inference

system. For the moment, each agent execution environment uses its own rules to encode and decode agents when they move from platform to platform. Furthermore, each platform still uses a different way for structuring agents and realizing their interaction. A need for common interaction and communication protocols has already arisen.

In the field of distributed artificial intelligence (DAI), considerable effort is pursued in the framework of the ARPA Knowledge Sharing Effort, which aims at providing interoperation between different kinds of heterogeneous agents. Perhaps the work being done there can also be applied to heterogeneous mobile agents. The ARPA Knowledge Sharing Effort includes the following three parts:

- **Ontolingua:** Intended for creating and maintaining vocabularies, it is important for ensuring that no confusion can arise when agents use different vocabularies as they exchange knowledge.
- **Knowledge Interchange Format (KIF):** A knowledge representation language proposed as a standard for interchange between other languages.
- **Knowledge Query and Manipulation Language (KQML):** A high-level protocol for interaction and communication between agents.

Another effort is being pursued in the arena of object-oriented programming in the framework of the Object Management Group's Common Object Request Broker Architecture (CORBA/OMG). CORBA aims at providing interaction between objects developed in different object-oriented programming languages. An object-oriented agent programming language can, therefore, benefit from the work done in CORBA.

## Mobility Support in Agent Languages and Environments

Both the advantages and the disadvantages of mobile agent languages stem from using scripts, even if previously precompiled, which must be downloaded and then interpreted, rather than simply executed as machine code. The application becomes architecture-independent because the same code can run on any host if the right interpreter is present on that host. The fact that distributed applications are independent from the hosts at which they are developed and executed makes application portability and upgrades easier.

The capability of dynamically downloading code at runtime enables dynamic extensibility of a distributed application, as new modules can be added to the application at any moment. As with late binding in object-oriented programming, the protocol to use in a data exchange between two hosts does not have to be chosen until the communication starts. This leads to an increased inter-operability, because the application can use any communication protocol, even one that is not wired in the platform, and the application can then interoperate with any other application in the network. This leads also to an increased flexibility, since new functionalities, as well as their desired implementations, can be added on demand.

Performance can increase with mobile code because the code can move closer to the data and can perform computations locally rather than remotely. One reason is that the code moves once across the network, and all subsequent computations occur locally without any more traveling over the network. Another reason is that sometimes it is unfeasible to move data to another site to perform calculations on it, especially when a great amount of data is involved in computing a small result. In those cases, moving data would increase the network traffic and bog down the computer's memory, all for a possibly poor result. Moving the code to the data instead is the equivalent of "going to the mountain" rather than moving the mountain to you.

Just as there is more freedom for using communication protocols, there is also more freedom for structuring agents. It becomes possible with the same language to structure agents either with the Remote Procedure Call (RPC) paradigm or with true mobility.

Unfortunately, using mobile agents brings not only advantages but also some crucial problems. As agents are usually interpreted, there are two problems that arise immediately: security (both hosts and agents need to be protected from each other) and efficiency (slow execution due to interpretation).

Several agent programming languages and their associated agent execution environments have been (and are still being) developed. These languages and execution environments deal with mobility in various ways and offer different agent structuring. In the following section, we briefly present agent mobility and structuring at both the language and the execution environment levels.

## Mobility at the Language Level

Agent programming languages offer to programmers different ways of expressing both agent mobility and agent behavior, and of realizing inter-agent communication.

### Primitives for Mobility and Spawning

Depending on the language and its distribution model, a sort of move primitive may not always be available. The presence or absence of this primitive can indicate the degree of true agency that the language offers. In the case where this primitive exists, it takes different names and different forms. For example,

Tcl and TACOMA	meet for moving and communicating with another agent
M0	submit for spawning itself or another messenger
Oblig	cloning and aliasing for placing copies of an object in multiple locations
Telescript	go for moving to another place, meet for interacting with another agent, and send for cloning copies of itself

An agent cannot only move itself but also populate the network with other agents by sending agents for execution to one or more sites. For example, an agent can spawn across the network a collection of agents that work for it and bring back their results.

The mobility primitives are closely related to the spawn one. Most of the time, an agent moves by creating another agent that inherits its state and behavior at the new location, then ending its own execution and beginning the execution of the newly created agent in its place. It looks as if the original agent moved to the remote place.

### Inter-Agent Communication

The agent execution environment is responsible for handling inter-agent communication. However, the agent language must provide the appropriate primitives for that purpose. As with inter-process communication realized by operating systems, there are multiple ways to achieve inter-agent communication in an agent execution environment.

These interactions can occur either synchronously or asynchronously; by letting each agent identify the partner explicitly or not; by message passing or by shared memory; and by the necessity or not to be in the same platform in order to interact. Using the diverse combinations we can create with these interactions, we can design a spectrum of inter-agent communication schemes going from a strong communication, in which every agent knows at what moment and with what agent it is communicating, to a more free communication, wherein the agent is not aware of when, where, and with whom the communication takes place.

Agents (or more precisely, objects) that communicate by invoking the methods of other objects exemplify synchronous message passing in which the other agent's identity must be known because its method has to be activated. The agents involved in the communication do not necessarily reside in the same location, since the method can be invoked remotely.

Agents that meet with Telescript or TACOMA are examples of agents communicating synchronously. They know the identity of the agent to contact, and the agents involved must be in the same place at the moment of the communication.

Actors of Agha, which communicate by letting one actor send messages to another actor specified by a given address, are an example of agents that communicate asynchronously with a well known agent by message passing. The agents do not have to be in the same actor space at the same time.

In M0, messengers communicate by depositing values in a shared memory and then retrieving values in this shared memory. They are an example of agents that communicate asynchronously and anonymously through a shared memory without being in the same host while communicating.

Agents can also organize the way they inform other agents of the services they are able to offer. This generally occurs through an intermediary, which collects both services' names and the way agents can access the services. Agents can then access this intermediary and the service itself or the agent that provides the service, using one of the inter-agent communication schemes listed earlier. Languages such as Phantom, Obliq or M0 offer the agents the possibility of publishing and retrieving services.

### Agent Mobility and Structuring of Agents

Primitives for mobility offered at the language level are usually only an interface through which the programmer interacts with the execution environment that actually implements the mobility. One can implement mobility in various ways at the execution-environment level, independently of the language primitives. Most of the time, agent structuring relies on the way mobility is handled.

As for inter-agent communication, it is possible to sketch a spectrum of designs for agent structuring, based on the way the structuring enables agents to move or to realize only transparent remote procedure calls.

At one end of the agent mobility spectrum, we find agents that do not move but perform their work by using remote procedure calls. Their execution always occurs at the same place, and they let only the results of remote computations travel the network to reach them.

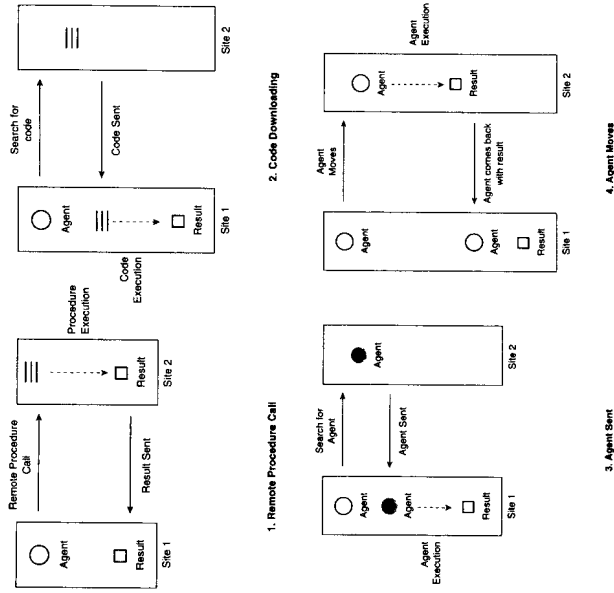
The next step consists of a simple downloading of code. Agents do not move but are able to let some pieces of code come to their execution site. The code executes locally, and the results become immediately and locally available. Generally, the code comes as a result of a remote procedure call. This method is relatively simple, as it happens without any state being kept and without the code itself being aware that it is executing at a new location.

At a further step, we find languages that enable objects to migrate together with their state and values, even though the objects do not explicitly migrate on their own desire but at another object's request. This is also called *process migration*, as the process is not aware that it has moved.

At the other end of the spectrum, we find self-desired migration of executing entities. The agents enclose their state and behavior, and they realize their proper migration. At this point, there is no entity requesting the migration of another entity in order to reach its location—there are only entities that decide themselves to migrate or entities that explicitly send other entities to other locations.

Figure 20.3 shows the four kinds of mobility.

**Figure 20.3.**  
*Four kinds of mobility.*



## Application Domains

This section presents some domains in which mobile agent technology can be immediately applied to build distributed applications. In some domains, mobile agent technology is attractive because it represents a new way of expressing distributed applications with mobile code. That is, applications become aware of distribution. In other domains, mobile agents can improve network usage by reducing data exchanged between hosts.

### Information Search and Retrieval

We believe that applications that search and retrieve information in some form will be among the first applications to benefit from the agent technology. In these kinds of applications, information usually resides on a number of servers and is accessed by clients using remote procedure calls.

The servers offer generic services to support different kinds of client applications. As a consequence, most of the information transferred to a client by a server is not meaningful for the client application, and the client application has to filter the received information. Using the agent technology, you can send an agent to move among the hosts containing the information.

The agent will search, retrieve, and process the data in the most appropriate way for the client application. Each application will, thus, have a kind of personalized server tailored to its own requirements.

The client application is even free to choose its interface with its personalized server. For example, a user can use requests in a natural language that will be directly forwarded to its agents. In an application where the amount of data to be transferred between hosts is very high compared to the size of the application code, such as in geographical databases in which terabytes of information are manipulated, moving code—using agent technology—results in more efficiency and flexibility than moving data.

### Active Mail/Document

Imagine a group of people working in different places needing to schedule a meeting. Instead of having a physical person play the role of the coordinator, you can have an agent collect all the people's availabilities and find the best day and the best hour for everyone.

One way to make this happen would be for the coordinator agent to send a mobile agent to each person as a calendar. The mobile agent waits for the user to choose his or her best day and time and roam back to the coordinator agent. Another solution is to have only one agent, which travels from one person to another one, displaying the calendar. The information on the calendar increases at each stop. Once it had completed its turnaround, it proposes a date and time and, if necessary, starts a new series of visits to the users to see if they agree with the proposal.

### Routing

Routing in a global network becomes more complex as the network grows. Traditional routing based on addressing schemes requires that a packet destination be explicitly named by the sender entity. Most of the time, the sender does not have enough information to control how the packet travels to its destination. It is here that a mobile agent can become handy. On its way to its destination, the agent can collect information to enrich its knowledge about the network topology, the network load, a possible network congestion, and so on.

Based on this information (which is not known when the agent is launched on the network) and on the gravity of service requested by the agent owner, the agent can decide for itself or can provide routing software hints on how it can reach its destination. Routing ceases to be static and becomes a dynamic process under which the information to be routed (the agent) interacts with the routing entity (the routing software). You can even imagine that the sender might express only a goal to reach, without providing the agent with explicit information on the location where it can find the necessary information to satisfy the goal. The agent would then roam in the network, trying at the same time to optimize network resource consumption, until it finds the information to complete its tasks.

## Network Management

Network management requires that entities trying to automate the management process act on important asynchronous events as soon as they occur. In some approaches, monitoring tools collect information about remote equipment and send it to a host, where an automatic management tool identifies the problem, its causes, and the appropriate actions to remedy it. Sometimes, it is difficult for the collected information to reach the decision-making host—for example, if the problem to be solved is network congestion due to overload by a misbehaved remote application. One might consider positioning mobile agents on important nodes, where they would react directly on local events, and from where they could move to remote nodes placed under their management control to handle remote events.

## Electronic Market

More and more business transactions will occur through the network, with a client and a vendor who do not physically meet. At each stage of a transaction, mobile agents can help. As there are more and more vendors offering a wide variety of products, a mobile agent can travel through this electronic market to find all the vendors offering the needed product and then propose to the client a list of the vendors offering the best price/quality rate. Once the client has narrowed down one or more vendors, mobile agents can then initiate transactions with the vendors, ironing out details such as delivery delay and payment facilities. When the user finally chooses the best vendor, mobile agents travel to all the other vendors to cancel the other transactions underway. A mobile agent then goes to the chosen vendor to finalize the contract.

The client's electronic signature can be encapsulated in an encrypted mobile agent that delivers it to the vendor. It is the same for the payment—an agent can deliver money from the client's account to the vendor's account.

## World Wide Web Browsers

Some graphical interfaces to the Web enable the user not only to view some static contents (pages), but also to view dynamic and interactive contents. Instead of a static page, the user receives code that executes on the user's host. This results in either dynamic contents that the user can visualize, or a graphical interface through which the user can interact and even modify data stored at the other end of the Net.

Mobile agents bring a new dimension to the Web. Each hypertext link that a user follows could send a mobile agent to the user that displays the correct application on the user's screen—a classical Web page, moving images, a pop-up window, an interactive question-and-answer form, and so on. In some cases, a hypertext link might not cause a mobile agent to come to the user; it might cause a mobile agent to go somewhere else for a given purpose, such as to activate a computation on a high-speed host.

Mobile agents can also enhance Web browsers to handle any protocol, image, or text format. An agent enabling the browser to perform the protocol correctly or to handle the new image or text could be sent together with the data exchanged, the image, or the text. Mobile agents could also be used to perform statistical computation on both the information accessed by users and on users themselves—their machines, their access locations, the interfaces they use, and so on.

## Commercial and Academic Products

This section presents, from a mobility point of view, some products, languages, and environments that can help build distributed applications using mobile agents.

There is currently a great interest in both the academic community and in the computer industry in environments that might be well suited for distributed mobile agent applications. The list in the following sections is not exhaustive. However, it presents some well-known environments such as Java or Telescript from an agent mobility point of view, together with some emerging environments like M0, Obliq, Phantom, and TACOMA that provide some interesting features for mobile agents.

### Java

The Java language environment is a Sun product that enables programmers to build applications that will run in a distributed environment on heterogeneous and not necessarily compatible hardware architectures and operating systems. The Java language's major application is the HotJava Web browser, a tool that increases flexibility in the use of the Web at both the content and protocol levels.

### Java Language Environment

The Java language environment consists of several elements: the Java language, the Java compiler, the Java interpreter, and the Java code generator.

The Java language is an object-oriented language based heavily on C++, with some features removed (pointers and explicit memory management,) and some features added (as support for concurrency).

The Java language is an intermediate scripting-based language. A Java compiler produces an intermediate script, the Java bytecode, which is not machine code for a specified computer, but rather machine code for a virtual machine. It is completely independent of any operating system or computer architecture. It is the Java bytecode that moves across the network from one host to another.

A Java interpreter compatible with the host's operating system and hardware architecture implements a given host's virtual machine. Once the Java bytecode reaches a new host, it can (at runtime) be interpreted by the Java interpreter or turned into machine-native code for efficiency purposes by the Java code generator.

## The HotJava Web Browser

HotJava is the major application written in the Java language. HotJava is a Web browser that enables the user to view not only Web pages with static text and images but also pages with dynamic contents (images that change at runtime).

How does it function? HotJava supports an extension to the HTML language that understands not only hypertext links, but also links to Java bytecode. It can then read Java pages by downloading to the user's host not only the Java HTML-like code of the wanted page but also all the Java program bytecode (the *applets*) needed by the page. Once the bytecode has reached the user's host, the HotJava browser lets the user see the Web page and invokes the Java interpreter to execute any applets the user may request.

The user is able to see dynamic contents—for example, with an applet whose execution leads to a sequence of images. Another advantage of a scripting language is that no protocol needs to be wired in HotJava. A needed protocol reaches HotJava as a program, HotJava uses the protocol. In that way, the HotJava browser is able to use any protocol provided in the Java language, not just pre-installed protocols. Similarly, the HotJava browser can understand objects of a type previously unknown to it (for example, an image in a new format), as long as the Java code supporting the object comes along with the object.

## Security

The Java environment provides several layers of security. It uses a Java bytecode verifier to scan bytecode that has traveled the network and check its safety. It then maintains separate name spaces for local code and for code coming from elsewhere in the network. (There is one name space for each network source.) The HotJava browser enables some security policies concerning the file system and network accesses.

## Agent Mobility Point of View

The Java language provides a multithread programming style at the language level. An application can be built using multiple threads (or lightweight processes), executing concurrently. Monitors provide the threads with methods of starting, stopping, and synchronizing with other threads.

At the object level, we also find some agent-like interactions, as objects communicate using the message-passing paradigm. Messages are sent between objects by invoking methods. Objects have a state (the set of instance variables) and a behavior (the set of methods), and they support single inheritance.

Java code is moved from one point to another to increase the flexibility and interoperability of Java applications (for example, to enable the HotJava browser to understand a new protocol or to execute a dynamic content). Java code does not have the ability to decide to move itself from one place to another. For example, in the HotJava application, the Java code moves from one

place to another at a user's request. This code is not able to make its own decision to migrate to another location. However, some agent execution environments, based on Java, have succeeded in implementing mobility in which an agent decides itself to move.

## M0 and Messengers

The messenger paradigm was born in the computer communications arena at the University of Geneva, Switzerland. It advocates the exchange of computer programs, rather than predefined messages, between two communicating hosts. The programs exchanged between the hosts are called *messengers*. They contain not only the data being exchanged but also the rules for handling the data.

Using the messenger paradigm, communication between two computers ceases to be an interactive process, as proposed by Shannon's classical model of communication, and becomes an instructional process. Indeed, the host initiating the communication can choose a protocol for data exchange not known beforehand by its partner. It then sends the protocol's logic to the destination partner, and the destination partner can use the chosen protocol as though it had learned it during the communication process. The main consequence of this ability is that the two hosts need not have identical preconfigured entities inside them in order to communicate.

The messenger paradigm is a new way of enabling computer communication that can be used to build any kind of distributed software. The paradigm is particularly well suited for building mobile agents.

In order to use the messenger paradigm for communication in a network of computers or for building mobile agent-based applications, each host must contain an environment for messenger execution called a *messenger platform*. M0 is an implementation of such a platform. Each platform contains a set of unreliable communication channels that link it to neighbor platforms. All the hosts must communicate only by exchanging messengers through their channels. They, therefore, have to use a common language for expressing messenger behavior, and they must share common rules for encoding messengers before sending them to the network and for decoding messengers received from the network.

The messenger paradigm requires that any messenger reaching a platform be unconditionally turned into an autonomous process or thread of execution and be executed. At any moment, a messenger executing in a platform can create another messenger for execution in the same or in a remote platform. The new messenger is completely independent from the messenger that created it. A messenger can also decide to move itself to another platform.

Messengers executing inside the same platform can synchronize their execution through process queues and exchange data only through a global store. The messenger language and its platform offer basic primitives for managing process queues (entering, leaving, stopping, and

restarting a process queue). Only the messenger at the head of a process queue is executable; all other messengers in the same queue must wait until they reach the head of the queue. When a messenger leaves a process queue (which is possible only for the messenger at the front of the queue), the other messengers move one step ahead in the process queue. Stopping a process queue freezes the messengers' progression inside the queue but does not affect the messenger executing at the head of the queue. Starting a process queue unfreezes the progression of messengers inside the queue.

In summary, messengers are anonymous and autonomous threads of control that can spawn a network of messenger platforms. *Anonymous* means that once a messenger has been successfully created, it can not be killed either by the platform or by another messenger. *Autonomous* means that a messenger decides by itself what to do, when to interact with other messengers, and where it should go in order to achieve its task.

### The M0 Messenger Language

The M0 language—pronounced “*M zero*,” for the first generation of Messenger languages—is the common language used to express messenger behavior in M0 platforms. It derives from PostScript. It is a postfix, stack-oriented, and interpreted language. All PostScript graphic-related operators have been removed, and the language has been extended to support concurrency.

### The M0 Messenger Platform

M0 is a messenger-platform prototype implemented at the University of Geneva to experiment with messengers and to study the impact of mobile code on distributed applications. The M0 software is available in the public domain.

Messengers execute concurrently inside a platform that is responsible for managing the resources available inside the host computer. M0 offers only basic local services to messengers. Services that require the cooperation between platforms are implemented by messengers themselves. Therefore, a messenger running in an M0 platform cannot directly access services available in another platform. The messenger has to move to the remote place to access services available there.

Although all M0 platforms on a network will offer basic services to messengers for their execution, messengers themselves will enrich their platforms with different services. The nature and complexity of services available in an M0 platform will vary from platform to platform. To allow messengers arriving in a platform to discover which services are available, messengers use

common conventions to publish the services they implement. These conventions enable service providers to publish their services securely and service consumers to locate services they need and use them appropriately.

The global store used to exchange data between messengers is managed as a set of two dictionaries storing associations between a key and a data item. A data item is accessed through its associated key. One of the global dictionaries is browsable (a messenger can obtain all the associations it contains) and is used by messengers to publish/locate services in a platform. The other dictionary is not browsable and is used for sharing data.

Messengers create random keys, which most of the time are random strings of 64 bits generated by the platform, and use them to store data in the global store. They distribute the keys to other messengers to which they thereby grant access to their data. A messenger can restrict access to its data, however, by distributing a read-only reference to it. The keys become a capability expressing the operations allowed on the associated data.

Different references expressing different capabilities can be created from the same key. A messenger can, therefore, grant different access capabilities to different messengers for the same data item. For example, some messengers would have read-write-execute access on a data item, while others would have read-only or execute-only access to the same data.

M0 manages the host computer's basic resources shared by all messengers. Resource management involves messenger scheduling, memory allocation to messengers on demand, and access to the network. A uniform market model is used for resource management. Messengers must pay for all resources they consume for their execution, using a nonforgeable currency. For example, lottery scheduling is used to manage the CPU time. Messengers pay tickets to participate in a lottery, and the lottery winner gets the CPU for a fixed time. Similarly, messengers sponsor their data resident in main memory. If different messengers share a data item, each can create an account to sponsor the shared data. A kind of offer-demand law determines the different resources' prices. Resources are expensive when their demand is high, and they become less expensive as their demand drops. Messengers that cannot pay for the resources that they need for their execution perish. This leads to a dynamic and self-adapting resource-management mechanism.

### Security

In M0, when a messenger reaches a platform, it is unconditionally executed. There is no attempt to determine the messenger's origin or to do any kind of authentication. M0 offers basic facilities that messengers can use to implement security policies suitable to their requirements.

These facilities include the capability to create random keys, to control access to shared data, and to use encryption functions to generate public keys from secret keys.

The choice not to implement a security policy at the platform level is motivated by the following facts:

- No single security policy can meet all messengers' requirements, and hence not all messengers will be willing to pay for it or incur the efficiency loss it induces.
- If messengers have to be used to conclude contracts for their users in an electronic market, anonymity and no access denial to platforms can be attractive.
- Enforcing a security policy at the platform level is very costly because it requires complex coordination between different platforms, but M0 aims at offering only basic local services to messengers that must implement services spawning more than one platform.

### Agent Mobility Point of View

Messengers can move across the network from platform to platform. They autonomously decide when, where, and what to move. The move occurs when a messenger executes the submit operation. Actually, the submit operation results in a new messenger's creation. The messenger executing the submit operation specifies explicitly

- The code to be executed by the new messenger
- The new messenger's initial state (data)
- The location (platform) where the new messenger will begin its execution

Therefore, a messenger moves itself to a remote location by executing the submit operation as its last instruction in its source platform. Because messengers do not have identity associated to them, their only important features are their codes, their states, and their locations.

### Obliq

Obliq is developed by Digital Equipment Corporation. It addresses more particularly the problem of application distribution.

### The Obliq Language

The Obliq language is an interpreted, untyped, and object-oriented language based on objects rather than classes. This is also called *prototype* or *embedded-based*, as opposed to *delegation-based*. This means that Obliq objects are self-contained—all methods and values the object needs are embedded in the object itself and are not shared with other objects.

Objects are the basic structuring entities in Obliq applications. An object consists of a set of values and a set of methods. Operations on objects are

- Invoking a value or an object's method
- Overriding a value or a method
- Cloning one or more objects
- Aliasing a value or an object's method
- Aliasing the object itself

If the invocation occurs locally, it returns the result of the method. If the invocation occurs on a remote object, arguments are transmitted across the network, the method executes remotely, and its result is sent back across the network. Note that the result of a method can be a procedure—it can be a way of downloading code.

It is the same for overriding a value or a method. If the overriding occurs on a remote object, the new value or the new method is sent across the network to the remote site.

Cloning an object results in a copy of the object. The clone and the original object do not necessarily reside in the same location. It is also possible to clone multiple objects; in this case, the clone is one object whose methods and values are the union of the cloned objects' methods and values.

The aliasing redirects a value, a method, or an entire object from one object to another.

There are two different kinds of migrating entities in the Obliq environment: procedures and objects. It is possible to transmit procedure codes (also called agents) together with all the bindings (or network location) to retrieve their values in the original location. These two pieces of information—the code and the set of bindings—are called the *closure* of the procedure. A procedure code is moved and executed in a location site that is not its original location site. All the values that the code needs are invoked or overridden at the original site. Because bindings give the address of the values' location, the code retrieves the values by following the bindings.

This way of transmitting procedure codes together with value locations is called *distributed lexical scoping*. Every free identifier appearing in the procedure's code is bound to some location. A disadvantage is that the information traffic increases because of the bindings and because it is necessary to follow the links to access data or methods.

Object migration is not as simple as procedure migration. Objects have a state, and it is necessary to maintain and retrieve the object's state while it is migrating. For this reason, Obliq does not allow an object to migrate—only references to an object migrate, with the object itself remaining in its original location. However, by the cloning and aliasing, it is possible to create a clone of the object at another location. The clone has the same state as the object itself, and it accesses the same values stored in the original location.

In addition to the cloning, an alias from the original object to the clone is added. This alias ensures that all methods are invoked directly on the original object. It actually invokes the clone's methods. It is the same for any operation performed on the original object—it is redirected to the clone site and executed locally at the clone site, as shown in Figure 20.4.

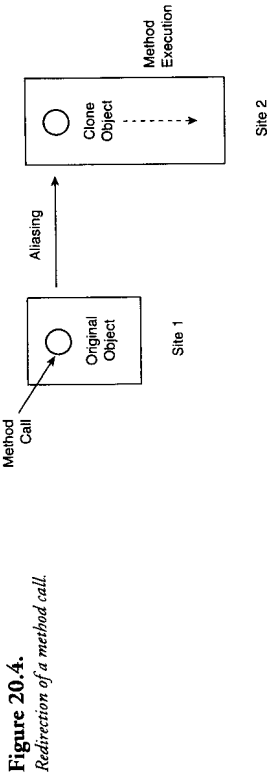


Figure 20.4. Redirection of a method call.

### The Obliq Runtime Environment

The Obliq environment supposes that we have interconnected sites, each containing an Obliq interpreter. Each site contains a collection of objects together with their values stored in the site under given locations. When a procedure closure transmits from one site to another, the code is actually transmitted, while only references to objects or values are transmitted. In this way, we obtain an interconnected set of objects and procedures, in which an object's procedures are executed at a site different from the object's original location, and in which the procedures' results are transmitted to the original location.

Communication among Obliq programs occurs by some programs publicizing the services they can offer and by other programs retrieving those services that want to use (see Figure 20.5). This happens by means of an external process, the name server, which maintains the associations between service names and their network location. The name server acts as an intermediary between servers (publishing the services) and clients (using the services). A (server) program lets its service become available to other programs by registering an object location under a given name with the name server. A (client) program retrieves the object location by using the name near the name server. The client program then accesses the object remotely, using its location, without needing the name server's help anymore.

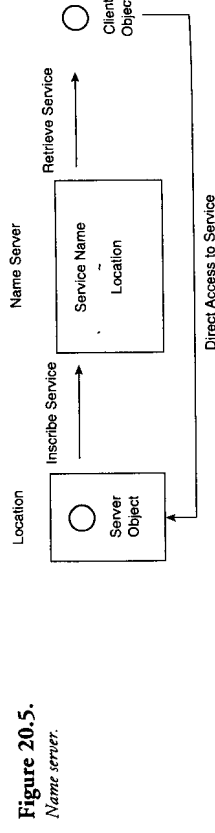


Figure 20.5. Name server.

### Security

Lexical scoping, used for updating and retrieving values at the originating site, is also used for security in Obliq. Obliq assumes that if all of a procedure's free identifiers are bound through the closure, there is no possibility for the procedure to retrieve or modify a private value either at the execution site or at the originating site. The developers plan to add other security features to the Obliq language as well.

You can protect objects to avoid external operations overriding their methods, for example. You can also serialize objects to prevent several threads (parts of some Obliq programs in execution, for example) from operating simultaneously on the object. An object is considered serialized if only one thread at a time can operate on the object and if a method of an object can call itself or another method of the same object without causing a deadlock. This happens through the notion of *mutex*, which enables one thread to take hold of the object and provides conditions that allow one thread, having acquired the mutex, to stop its execution and let another thread execute on the object. To avoid the deadlock of a method calling a method of the same object, only external operations are able to acquire the mutex—no internal operation can.

### Agent Mobility Point of View

The Obliq language's embedded-based nature enables easy object cloning and aliasing. There is no need to clone complementary objects other than the cloned one. The cloning and the aliasing make the object mobile. However, the clone is a copy, not really a movement of the object, although the result is the same as if the object had really moved. Indeed, the use of distributed lexical scoping, especially the collection of identifier bindings, creates its own network of bindings, and it enables transparent execution of remote methods as well as transparent access to data values.

Operations performed on an object can be initiated either by the object itself or by other objects. This means that an object can decide to put its clone in a given location and, thus, move to this location.

### Phantom

The Phantom language and runtime environment are being developed at the Department of Computer Science at Trinity College, Dublin. The Phantom project is intended for the development of distributed applications needing both remote network access and an interactive user interface (Web browsers, distributed conferencing systems, and so on).

### The Phantom Language

The Phantom language is an interpreted language derived from Modula-3, with some features added and others removed. Among other features, it contains modules and interfaces, procedures, objects with single inheritance, and thread types. It provides for transmitting

program codes from one site to another for execution at the destination site and transparent access to remote codes or data.

Objects in Phantom have a state (a set of variables) and a behavior (a set of methods), and support single inheritance. Due to the problem of state consistency, the programmer must explicitly call for object migration—an object cannot migrate on its own volition. Even when migration is called for explicitly, only a reference to the object migrates.

At the language level, it is possible to define concurrent threads of control. It is then possible to create threads that wait for the end of a thread's execution. Two threads synchronize their access to shared variables using both mutual exclusion semaphores and conditions. As for objects, threads are not transmitted from one site to another. Only references of threads are transmitted.

As we have seen, for state consistency, only references on objects and threads are actually transmitted across the network. The only program codes transmitted are the procedure codes. In a client-server example, the client can invoke a method of the server transparently and remotely. The result returned by this method can be a procedure, as Phantom allows high-order functions. After this procedure's code has been transmitted to the client side, the client can execute the procedure locally.

More precisely, not only is the procedure code transmitted, but also the closure of the procedure. The closure consists of the procedure code and the environment—all the free variables appearing in the procedure code together with their location addresses (where the procedure comes from). In that way, the variable's value is transparently and remotely accessed when the procedure needs it. Any change of a variable value is also available to the procedure, even if the procedure is being executed in another site.

The Phantom language belongs to both the scripting languages and to the intermediate-based scripting languages, as the procedure codes that are transmitted from site to site can either be source code text or some internal representation for a virtual machine.

Using the references on objects and threads, and the closure of procedures, the Phantom runtime environment makes transparent the execution of remote code and the access to remote data.

### The Phantom Runtime Environment

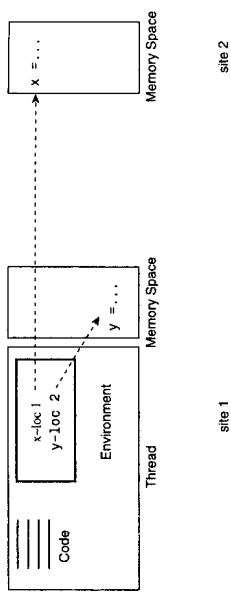
The Phantom distribution model is similar to the Obliq distribution model. It has the same notions of lexical scoping for free identifiers and thread synchronization.

The Phantom runtime environment is also called a *site*. It invokes the Phantom interpreter running on the host machine. Sites are interconnected through a network, and each site has a unique site address consisting of the host's IP address and a port number.

Each site's Phantom interpreter maintains a single memory space containing values uniquely identified by their location inside the memory space. Programs execute as multiple threads. Each thread is subdivided into an environment (mapping variables and global location

addresses) and a program code. According to the distributed lexical scoping, the environment enables every variable appearing in the program code to be retrieved from the network. It can be a variable whose value resides in the local site's memory space or a variable whose value has to be retrieved from another site's memory space (see Figure 20.6). The global location address specifies the site address and a location in its memory space. However, the program code could be the source code; it is currently not the source code, but instead a representation (bytecode) for a virtual machine.

**Figure 20.6.**  
*Distributed lexical scoping*



Phantom provides the same mechanism as does Obliq for enabling agents to publish and retrieve services by using a name server process that is dedicated to interface between publisher agents and retriever agents.

### Security

Lexical scoping is used to prevent received procedures from having access to more resources on the local site than an RPC from the remote site would access. The Phantom interpreter verifies that all the free variables appearing in a received procedure's code also appear in the environment sent together with the procedure.

The identity of the user who started a Phantom interpreter is maintained as an object by the interpreter. An authentication protocol based on public-key encryption tells an interpreter the identity of the user who started a remote interpreter with which it intends to communicate.

Private data can be safely stored in the memory space because it is stored under a key generated by the site possessing the data, rather than stored using a pointer that could be easily followed by a malicious program.

At the language level, it is possible to mention permissions—that is, access control—of a given object's methods. Users other than the owner of the object can be granted or denied permission to read, write, or execute the method. The user who has started the interpreter is considered the owner.

### Agent Mobility Point of View

Programs communicate through services. One makes a service available to other programs by registering an object under a server name and by mapping between the server name and the object's global location address.

Only procedures may be mobile—objects and threads are accessed by remotely executed references and methods. Transmitted procedure codes are called *agents* in the Phantom framework, although it consists only of code downloading. As in other languages, mobility occurs at a program's request, and not under an object, thread, or procedure's own initiative.

Phantom uses the same distribution model as Obliq for the lexical scoping of the free identifiers of transmitted procedures, the synchronization of threads, and the use of a name server. However, Obliq and Phantom differ on other points, such as the notion of objects. Obliq is based on objects (or prototypes), whereas Phantom is based on classes. This leads to some differing ways of managing mobility.

Both transmit procedure closures across the network and use the procedure identifiers' binding to retrieve values across the network. In both cases, only references to objects are transmitted, not the objects themselves. However, because Obliq uses self-contained objects rather than classes, it can go a step further by cloning objects at different sites, effecting a kind of object mobility.

## TACOMA

The TACOMA (Tromso And COrnell Moving Agents) project is being developed at the Tromso University in Norway and at Cornell University in the United States. The project's main focus is both on operating systems for agent-based computing and on structuring agent applications in a distributed environment.

### The TACOMA Language

TACOMA version 1.0 derives from Tcl-TCP (Tcl with TCP network communication) and is based on UNIX.

The basic unit of computation is the *agent*. An agent consists of code and a state (the data); it is a simple process in execution. However, you can build any complex application using this basic notion of agents. Agents are written in Tcl, but any piece of code in any language can be moved and executed at any site. The only requirement is that a compiler or interpreter exists for that language in the site.

There are no more notions of client and server, but only the notion of agent. Agents can both send code and receive and execute it. For example, agents can decide to continue their execution at another site or to parallelize their execution by sending one or more agents to different sites for execution.

### The TACOMA Model of Computation

The TACOMA approach realizes both the true mobility of agents and the mobility of agents driven by the agents themselves. By "true mobility" we mean that an agent (its code and current state) actually moves from one site to another one, where it resumes its execution. The few notions of folders, briefcase, file cabinet, and particularly the meet instruction allow TACOMA agents to move as they want.

When an agent travels across the network, it is necessary for it to travel with its data (or state) and its code. Data is organized into *folders*, which agent can access to store and retrieve their data. Different folders are available: folders for data values, folders for the agent's code, folders for the host to which the agent wants to travel, folders for the remote agent to contact once the agent arrives at the remote site, and so on.

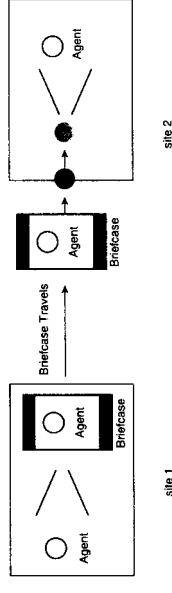
The agent travels the network with one or more folders, stored together in what is called a *briefcase*. The briefcase moves from site to site, containing all the information necessary to move the agent to the right location, to retrieve its code and state, and to let the agent meet the contact agent at the remote site. An agent does not necessarily travel with all the information; it takes with it only the folders it needs. Folders remaining at the original location are stored in the *file cabinet*.

An agent can move only if it wants to meet some other agent—the *contact agent*—at another site. The meet instruction sends the briefcase to a host to make contact with the contact agent.

Once a briefcase reaches the new host, a special firewall agent verifies some security features of the briefcase and then lets another special agent unpack the briefcase to retrieve both the code that has to be executed (the traveled agent) and the name of the local contact agent that the traveled agent is to meet. The actual meeting consists of delivering one or more folders to the contact agent (see Figure 20.7). The contact agent can be very simple and is used only to enable the traveled agent to reach the new location and to execute.

Figure 20.7.

An agent moves by enclosing itself in a briefcase.



## Security

The TACOMA environment ensures security. Before an arriving briefcase enters a site, the firewall agent checks it accepting it only if it passes inspection. The firewall agent performs operations such as authentication and access control. It also realizes accounting.

### Agent Mobility Point of View

TACOMA agents actually move across the network on their own desire. An agent can pack its own code and data in a briefcase and then execute a meet. It travels the network (actually a copy of itself travels) and resumes its execution at the new location.

The meet instruction moves a briefcase from one place to another and also communicates with agents at the same site. During the meet, agents exchange folders. TACOMA agents meet only if they are at the same site. They do not exchange messages remotely from one site to another.

TACOMA agents decide for themselves what to do with their computation. They freely structure their execution by executing themselves from site to site, by using the RPC paradigm, by parallelizing their execution, or by sending an agent to a given site for computation while waiting for its execution.

With TACOMA agents, the situation is the reverse of what we find in the client/server paradigm. In the client/server paradigm, the server waits for a client to request, for example, a code. In TACOMA, there are neither clients nor servers. Each agent can play any role. For this reason, not only can a piece of code be sent on request, but an agent can take the initiative to explore the network by sending its code.

### Telescript

Telescript, developed by General Magic and supported by a number of companies, aims at becoming an enabling technology for electronic commerce over a global network such as the Internet. For more information about Telescript, see Chapter 7.

### The Telescript Language

The Telescript language is an object-oriented language designed for programming distributed applications using the mobile agent technology. The source code is compiled in an intermediary bytecode for a virtual machine and is interpreted by a Telescript engine place. When you build an application, you can write parts of it in a traditional programming language and the moving parts in the Telescript language. By using permits, the language enables you to control the capabilities granted to processes. In this manner, a process can be prevented from unduly manipulating its hosts or other processes. All Telescript objects are persistent; if an engine place fails, the processes can resume their execution, with their state restored, after engine place recovery.

### The Telescript Runtime Environment

The network of all hosts is called the *telescope*. It is divided for operational purposes into regions, each one under the operational and administrative control of a well-known physical person or organization called the *region authority*. All the computers and network devices inside a region are thus under the control of the region authority.

A region can be composed, at one extreme, of only one personal computer owned by an individual (for instance, using a wireless or modem connection to access the network). At the other extreme, a region can group a number of local area networks owned by an organization.

Each computer in the network contains a *Telescript engine place*, software responsible for executing Telescript processes. The engine place mediates the interaction between the host's operating system and the Telescript processes. The engine place is thus responsible for managing the different resources available in a host, such as CPU time, memory, network bandwidth, and so on.

The engine place controls the processes' execution and interactions. Telescript processes come in two flavors: *places* and *agents*. In the Telescript model, each process has a telename that identifies the process and its authority. A telename can also identify a set of processes (agents or places) belonging to a given authority. A process should not be able to falsify its telename or withhold it from other processes. A place has also a *teleadress* that locates it in the telescope and reveals the authority that operates the computer in which the place exists. As with telenames, a single teleadress can identify a set of places belonging to a given authority.

A place is a static process that offers a venue for other processes. One or more places or agents can occupy it. Places inside a host have a hierarchical structure, with the engine place at the top of the hierarchy. Next come places that are direct subplaces of the engine place, and next the direct subplaces of the direct subplaces of the engine place, and so on. Typically, the host's authority owns places inside a host, and provides a set of well-known services to agents. Different agents belonging to different authorities visit a place when they need to access a service provided by the place, or when they need to interact in that place.

### Security

In Telescript, security is enforced by both the language, which aims to be a safe language, and the execution environment, which is responsible for authentication and user authorization. For security purposes, each Telescript process (agents and places) has a unique, nonforgeable identity (telename) and a unique, nonforgeable owner used for authentication and user authorization. Permits grant particular capabilities to particular processes on particular occasions. This allows the system to control the access and resource consumption.

### Agent Mobility Point of View

A Telescript agent always executes inside a place. An agent moves from place to place, accessing the necessary services and collecting the information it needs to achieve its task.

A move occurs when an agent executes the go operation. For that purpose, the agent must present a ticket that defines the trip that it wants to take to the engine place of the computer inside which it is executing. The ticket identifies the agent's destination, which may be a place with a given name, any place of a given authority, any place at a given teleadress, or any place in a

given region, with a potential combination of some of these constraints. The engine place encodes the agent using a uniform encoding convention shared by all the engines inside the telephere, searches the places that satisfy the agent's ticket, and dispatches the agent to one of them, typically through the network.

When the agent arrives at the destination computer, if the dispatch was through the network, the destination engine place decodes the agent and presents it to its destination place. If the go operation succeeds (that is, if the destination place accepts the agent), the agent's next instruction executes in the agent's new location. The go operation can fail if the source engine place cannot locate a place that satisfies the agent's ticket, or if none of the candidate places accepts the agent. In that case, the agent either is left in its source location or is moved to a third unspecified location.

An agent can also create one or more clones of itself for execution in one or multiple places, with the send operation. The resulting agents are identical to their parent except for their identity and perhaps their capability to access resources available in the computer inside which they are executing.

Two agents can interact only if they are executing inside the same place. The two agents must agree to the interaction by performing the meeting operation. If the meeting operation succeeds, each agent receives a reference to the other, enabling them to access data owned by the other, provided that they have sufficient privileges.

## Summary

Agents have been known for a long time in computer science, but with the advent of networks they have received increased attention. The need for them has grown as the quantity of networked resources has increased exponentially. Only recently it became thinkable to be more aggressive and, instead of letting the agents sit at one location, to unleash them into the network. This new thinking has resulted in a quality leap, enabling the development of new network applications that would have been impossible or very hard to build within the classic computer communications approach. The main theme of mobility is that it makes sense to let the code go to the data instead of transferring data to immobile code locations.

However, although there may be agreement that we are witnessing a small revolution, little agreement can be found on how this revolution should happen. This chapter gave an overview of the divergent opinions on the questions of how mobility should be implemented at the technical level. Currently we are passing through a period of trials, in which development teams try to formulate a coherent technology. Due to market pressure, some teams take shortcuts and limit agent mobility to the areas where it can easily be mastered now. Other teams try to push the concept of mobile code as far as possible to have an advantage in the future. The shake-down phase has not begun yet.

Agent mobility alone is worthless without an understanding of how to exploit it. In addition to the problems at the infrastructure level, we urgently need to know more about behavior in the land of mobile agents. Designing and programming mobile agents already is and will become an even more demanding task because in a networked environment—nothing remains as it is. An open network for agents will mean that mobile agents will be exposed to rude competition. This is where intelligent agents and mobile code must meet: Mobile code must become adaptive and intelligent, and intelligent agents must become mobile.

## Other Resources

- Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press.
- Baumann, J. (1995). "SIDAS: The Simple Distributed Multi-Agent System—A Short Overview." ECOOP'95 Workshop W10, Aarhus, Denmark.
- Cardelli, L. (1995). "A Language with Distributed Scope." *Computing Systems* 8(1), 27–59.
- Chess, D., et al. (1995). "Itinerant Agents for Mobile Computing." IBM Research Report RC 20010.
- Ciancarini, P. (1993). "Distributed Programming with Logic Tuple-Spaces." University of Bologna Technical Report UBLC5-93-7.
- Courtney, A. (1995). "Phantom: An Interpreted Language for Distributed Programming." Usenix Conference on Object-Oriented Technologies (COOTS).
- (1995). "Phantom: A Language Overview." (<http://apocalypse.org/pub/u/antony/phantom/overview.ps>)
- (1995). "Phantom: Language Reference Manual." (<http://apocalypse.org/pub/u/antony/phantom/refman.ps>)
- Di Marzo, G., et al. (1995). "The Messenger Paradigm and Its Impact on Distributed Systems." ICC'95 International Workshop on Intelligent Computer Communication, Cluj-Napoca, Romania, 79-94. (<http://cuswww.unige.ch/tios/msgr/home.html>)
- Finin, T., et al. (1994). "KQML as an Agent Communication Language." *Proceedings of the Third International Conference on Information and Knowledge Management*. ACM Press.
- General Magic, Inc. (1995). *Telescript Language Reference*.
- Genereseth, M.R., and R.E. Fikes (1992). "Knowledge Interchange Format Version 3.0 Reference Manual." Stanford University Technical Report Logic-92-1.
- Goosling, J., and H. McGilton (1995). "The Java Language Environment: A White Paper." Sun Microsystems. (<ftp://ftp.javasoft.com/docs/whitepaper.A4.ps.tar.z>)
- Gruher, T.R. "Ontolingua: A Mechanism to Support Portable Ontologies." (<http://www-ksl.stanford.edu/knowledge-sharing/ontolingua/>)
- Harrison, C.G. (1995). "Smart Networks and Intelligent Agents." *Mediacom 95*.

- Harrison, C.G., et al. (1995). "Mobile Agents: Are They a Good Idea?" IBM Research Report.
- Johansen, D., et al. (1995). "An Introduction to the TACOMA Distributed System." Tromsø University, Computer Science Technical Report 95-23.
- Lingnau, A., and O. Drobnik. "An Infrastructure for Mobile Agents: Requirements and Architecture." (<ftp://ftp.km.informatik.uni-frankfurt.de/pub/papers/agents/13dis-paper.ps.gz>)
- Merz, M., and W. Lamersdorf (1996). "Agents, Services, and Electronic Markets: How Do They Integrate?" IFIP/IEEE International Conference on Distributed Platforms, Dresden.
- Mowbray, T.J., and R. Zahavi (1995). *The Essential CORBA: Systems Integration Using Distributed Objects*. Wiley.
- Spafford, E.H. (1989). "The Internet Worm Program: An Analysis." *ACM SIGCOMM Computer Communication Review*, 19(1), 17-59.
- Sun Microsystems (1995). "HotJava: The Security Story." Sun Microsystems. (<http://java.sun.com/1.0alpha3/security/security.html>)
- (1995). "The HotJava Browser—A White Paper." (<http://java.sun.com/1.0alpha3/doc/overview/hotjava/browser-whitepaper.ps>)
- Tschudin, C.F. (1994). "An Introduction to the M0 Messenger Language." University of Geneva technical report. Cahier du CUI No. 86. (<http://cuiwww.unige.ch/tios/msgr/home.html>)
- (1994). "M0—A Messenger Execution Environment." Usenet Newsgroup comp.sources.unix, vol. 28, issue 51-62. (<file://cui.unige.ch/pub/m0>)
- White, J.E. (1994). "Telescript Technology: The Foundation for the Electronic Marketplace." General Magic white paper.