Designing Distributed Applications with Mobile Code Paradigms

Antonio Carzaniga
Politecnico di Milano
Piazza Leonardo da Vinci, 32
20133 Milano, Italy
+39-2-2399-3638
carzaniga@elet.polimi.it

Gian Pietro Picco
Politecnico di Torino
Corso Duca degli Abruzzi, 24
10129 Torino, Italy
+39-11-564-7008
picco@athena.polito.it

Giovanni Vigna
Politecnico di Milano
Piazza Leonardo da Vinci, 32
20133 Milano, Italy
+39-2-2399-3666
vigna@elet.polimi.it

ABSTRACT

Large scale distributed systems are becoming of paramount importance, due to the evolution of technology and to the interest of market. Their development, however, is not yet supported by a sound technological and methodological background, as the results developed for small size distributed systems often do not scale up. Recently, *mobile code languages* (MCLs) have been proposed as a technological answer to the problem. In this work, we abstract away from the details of these languages by deriving design paradigms exploiting code mobility that are independent of any particular technology. We present such design paradigms, together with a discussion of their features, their application domain, and some hints about the selection of the correct paradigm for a given distributed application.

Keywords

Mobile code, design paradigms, distributed applications.

INTRODUCTION

Distributed systems have been investigated for years, but recently research on this subject has gained a new impetus, partially due to the explosive growth of the Internet. Being the largest distributed system ever built, Internet is making distributed systems available to the general public, taking advantage of both the achievements in network technology and the ever-increasing interest of markets in long-haul communications. Consequently, a great deal of research is focusing on the exploitation of new broadband communication devices and in the provision of new services on large scale distributed systems, such as the Internet.

The technological and methodological background developed for conventional distributed systems often fails to scale up when applied to the problem of providing large scale distribution. As often happens, researchers tried to overcome the problem with a bottom-up approach, by developing technology providing *ad hoc* support for this kind of systems.

One of the outcomes of this research is the recent proposal of a number of new programming languages, especially conceived for the Internet, that are all characterized by the capability to provide some sort of *code mobility*, i.e., the capability to reconfigure dynamically, at run-time, the binding between the software components of the application and their physical location within a computer network. Work on code mobility is not new; other approaches [3, 8] already investigated the capability to provide code mobility at the language level. The novelty of the new approaches is in their emphasis on the application of code mobility to a large scale setting.

In our view, good technology addresses only a part of the problem. Software engineering taught us that a good software product does not come just from technology. Rather, higher level phases in the development process, such as specification and design, play a central role in the final success [1, 16]. Research on distributed systems partially fails to address these issues within a well-established framework able to guide an engineer through the development of a distributed application, and this deficiency is even more evident when large scale systems are involved.

In this paper, we are concerned mainly with the design of distributed applications, that aims at identifying the distributable components and their interactions which together satisfy the system requirements [10]. Our goal is to develop a repertoire of design paradigms [15] that can be used to design distributed applications exploiting code mobility. We achieve this by leveraging off of the experience we gained in our ongoing research [4] about programming languages supporting code mobility, often called *mobile code languages* (MCLs). We proceed bottom-up, trying to abstract away from MCLs and conceptualize the design paradigms they embody. The resulting paradigms are quite general and independent of the specific technology one may choose for the implementation although, of course, each design paradigm can be implemented more naturally by languages providing specific features [6]. Independence of the design paradigm from the underlying technology is a major point in the general software engineering practice. As an example, the benefits of an object-oriented design do not stem from the availability of an object-oriented language to be used in the implementation. Rather, they are valuable *per se*, for example in terms of improved understandability and reusability of the design.

The following section gives an overview of currently available MCLs, classifying them according to the degree of mobility they provide. Then, we give the rationale for mobile code design paradigms, followed by their definition and explanation. Subsequently, we discuss why and when mobile code paradigms can be beneficial and we tackle the problem of choosing a particular paradigm via analysis performed at the design stage. The final section illustrates further directions for this work.

A BRIEF OVERVIEW OF MOBILE CODE TECHNOLOGY

The interest about code mobility has been raised mainly by a new family of programming languages, usually referred to as *mobile code languages* (MCLs), that were proposed recently[1]. Although languages belonging to this family are being developed both in industry and academia, marketing forces are playing a fundamental role in the acceptance of such languages in the community of potential users. This is proven by the fact that two of the most well-known MCLs, that is, Java by Sun Microsystems [18], and Telescript by General Magic [11], come from industry.

Although both Java and Telescript are considered MCLs, they are at the extremes of a continuum representing the degree of code mobility that can be provided by a programming language. Telescript provides a sophisticated form of thread migration, in which a thread running on an interpreter (or *engine*, in the Telescript jargon) is able to migrate autonomously to a different engine, possibly run by a different machine, by executing a `go` operation. This operation causes the engine to suspend the thread, serialize it (together with its state), and transmit it to the destination engine specified as argument. There, it will be unserialized, and its execution will be resumed from the instruction following the `go`. Telescript is a very rich language, in which support for mobility of both the code and the associated state has been a driving factor for the whole language design.

On the other hand, in Java there is no support for migration of executing code. The only language feature that can be directly exploited for code mobility is provided by a programmable *class loader*. Instead of having a fixed policy for resolving class names at run-time, Java provides the capability to program where the code corresponding to an unresolved class name has to be retrieved. Hence, Java programs can download code from the net and link it dynamically at run-time.

Between these two extremes there is a full spectrum of MCLs that provide different degrees of mobility for their *execution units*[2] (EU). Currently available MCLs provide support for at least one of the following:

Strong mobility is the ability of an MCL (called *strong MCL*) to allow EUs to move their code and execution state to a different site. Executing units are suspended, transmitted to the destination site, and resumed there. For instance, Telescript is a strong MCL.

Weak mobility is the ability of an MCL (called *weak MCL*) to allow an EU in a site to be bound dynamically to code coming from a different site. This encompasses two cases: either the EU dynamically links code downloaded from the network (as can be done in Java), or the EU receives its code from another EU. In the latter case, two more options are possible. Either the EU in the destination site is created from scratch to run the incoming code, or a pre-existing EU links the incoming code dynamically and executes it.

Strong MCLs are a minority, because of the challenging problems they pose in both defining and implementing the semantics of strong mobility. Besides Telescript, the family of strong MCLs includes languages such as Tycoon, Agent Tcl and Emerald. Tycoon [12] provides thread migration like Telescript, but lacks Telescript's richness of abstractions and features. Agent Tcl [7] provides strong mobility where the whole image of the interpreter can be transferred to a different site by executing a `jump` instruction. Agent Tcl also supports weak mobility, as it provides a `submit` instruction that allows transmission of Tcl [14] procedures, together with a portion of their global environment, to an interpreter running on a different site. Emerald [3] also provides a form of strong mobility, though it is not completely under programmer's control.

Weak mobility has been around for quite a long time under many shapes. For example, the well-known UNIX `sh` shell language together with the run-time support

[1] For simplicity, we refer to the technologies mentioned here as "languages", even when they are extensions of existing languages. Furthermore, we will describe only briefly problems and issues in MCLs. An in-depth discussion is provided in [4].

[2] Hereafter, we will use the term *execution unit* to refer to the run-time representation of a single flow of computation, e.g., a UNIX process or a thread in a multithreaded environment.

provided by the `rshd` daemon can be regarded as a language implementation supporting weak mobility, in that it allows a user to send a shell script to be executed on a remote machine. In addition, the REV [8] system, that inspired much of the work on code mobility, provides an extension to remote procedure call (RPC) that allows one to send both the actual parameters and the code for a procedure to be executed on a remote machine. More recent approaches have been inspired by applications development on the Internet. As we mentioned, the Java class loader allows the programmer to customize the resolution of a class name, possibly downloading the corresponding code from any machine on the network. An extension of Java, called MOLE [17], adds the capability of sending code to another Java interpreter without modifying the Java interpreter. TACOMA [9] is an extension of Tcl that allows the programmer to send code together with any kind of data to a remote machine, in order to be executed there. Facile [5] is a higher-order functional language conceived for concurrent distributed programming where code mobility is naturally achieved because Facile functions are first class elements of the language. `MO` [19] differs from the above languages in that it is not conceived to be the language in which mobile code applications are programmed. Rather, its goal is to provide a middleware layer supporting code mobility for higher level layers.

DISTRIBUTED APPLICATIONS AND CODE MOBILITY

When designing the architecture of a distributed application, interaction among the various components is usually considered independent of the components' location. The location of components is simply regarded as an implementation detail. In some cases, such details are explicitly stated by the programmer in the implementation stage. In other cases, they are automatically defined by some middleware layer. For example, CORBA [13] intentionally hides the location of components to the programmer. In this framework, there is no distinction between interaction involving components residing on the same host and components residing on different hosts of a computer network.

This, however, is not the only possible approach to the design of distributed applications. There are cases when the concepts of location, binding of computational resources to locations, and migration to different locations need to be taken into account during the design stage. In some cases, the interaction among components residing on the same host is remarkably different with respect to the case where components reside on different hosts of a computer network, in terms of latency, access to memory, partial failure, and concurrency. As stated in [20], hiding such differences can lead to unexpected performance and reliability problems.

In this paper, we address the class of applications for which the concepts of location and mobility are so important that they affect the conceptual structure of the application as it is conceived in the design stage. The next section illustrates a number of design paradigms that emerged from our research. Such paradigms are general and provide a repertoire of architectural patterns that can be used to design distributed mobile applications in a systematic fashion.

MOBILE CODE PARADIGMS

In this section we will abstract away from the specific features provided by the language in which a distributed application is written. Our goal is to identify design paradigms encompassing code mobility which can provide guidance for the design of distributed applications.

Design paradigms will be defined in terms of the following basic abstractions:

Components are the composing elements of an architecture. They can be further divided into:

- ***Resource components*** embody architectural elements representing passive data or physical devices, e.g., a file, a network device driver, or a printer driver. A particular kind of resource is represented by *code components* which contain the know-how necessary for the execution of a particular task.
- ***Computational components*** embody a flow of control. An example is a process, or a thread. They are characterized by a state, which includes private data, the state of their execution, and bindings to other components, in particular to code components and resource components.

Interactions are events that involve two or more components. For example, a message exchanged between two computational components can be regarded as an interaction between them.

Sites are execution environments; they host components and provide support for the execution of computational components. In our paradigms, sites embody the intuitive notion of location. Hence, interactions among components residing in the same site are considered less expensive than interactions taking place among components located in different sites.

Implicitly, we assume the existence of an underlying network which provides support for all the communication facilities.

We will present our design paradigms in terms of interactions patterns that define the coordination and relocation of components needed to perform a service. To this end, we consider a computational component A, located at a site S_A, that needs the results of the computation of a service. We assume the existence of another site S_B, which will be involved in the delivery of the service. In order to obtain the service results, A starts the interaction pattern that leads to service delivery. Service execution involves a set of resources, the know-how about the service (its code), and a computational component responsible for the execution of the code. In order to accomplish the service, these elements must be present at one site *at the same time*.

In this context, we identify three main design paradigms that extend the well-known *client-server* paradigm to exploit code mobility. We will call them: *remote evaluation*, *code on demand*, and *mobile agent*. We distinguish the design paradigms according to the location of the different components before and after the execution of the service, the computational component that is responsible to execute the code, and where the computation actually takes place (see Table 1).

The presentation of the paradigms is based on a real life scenario where two friends—Louise and Christine—interact and cooperate to make a chocolate cake. In order to make the cake (the results of a service), a recipe is needed (the know-how about the service), as well as the ingredients (the resources that can be moved), an oven to bake the cake (a resource that can hardly be moved), and a person to mix the ingredients following the recipe (a computational component responsible for the execution of the code). To prepare the cake (to execute the service) all these elements must be co-located in the same home (site). In the following, Louise will play the role of component A, i.e., she is the initiator of the interaction, and the one interested in its effects.

Client-Server (CS)

> *Louise would like to have a chocolate cake, but she doesn't know the recipe, and she has at home neither the required ingredients nor an oven. Fortunately, she knows that her friend Christine knows how to make a chocolate cake, and that she owns everything needed at her place. Since Christine is usually quite happy to prepare cakes on request, Louise phones her asking: "Please, can you make me a chocolate cake?". Christine makes the chocolate cake and delivers it back to Louise.*

The client-server paradigm is well-known and widely used. An example is the X Windows system. In this case, the server manages a physical display while client applications use the display through the services provided by the server. For instance, one client may request the server to draw a filled rectangle passing the coordinates of the upper-left and lower-right corners. As a consequence, the server executes the procedure that actually draws the rectangle driving the physical display.

In this paradigm, a computational component B (the server) offering a set of services is placed at site S_B. Resources and know-how needed for service execution are hosted by site S_B as well.

The client component A, located on S_A, requests the execution of a service with an interaction with the server component B. As a response, B performs the service requested by executing the corresponding know-how and accessing the involved resources co-located with B. In general, the service produces some sort of result that will be delivered back to the client with an additional interaction.

Actually, a server may rely on other components in order to perform parts of the required service or to retrieve parts of the required data, but, in this case, the server would act as a client in another client-server interaction. From the original client's viewpoint the server owns all necessary data and knowledge.

Remote Evaluation (REV)

> *Louise wants to prepare a chocolate cake. She knows the recipe but she has at home neither the required ingredients nor an oven. Her friend Christine has both at her place, yet she doesn't know how to make a chocolate cake. However, Louise knows that Christine is happy to try new recipes, therefore she phones Christine asking: "Can you make me a chocolate cake? Here is the recipe: take three eggs...". Christine prepares the chocolate cake following Louise's recipe and delivers it back to her.*

There are several examples of a *remote evaluation* design paradigm[3] implemented using the available technology. For example, in the UNIX world, the `rsh` command allows a user to have some script code executed on a remote host. Another example is the interaction between an application (e.g., a word-processor) and a PostScript printer. The resources involved in this interaction are the printing devices (e.g., laser raster, paper tractor, and so on), while the code is the PostScript file, which is executed by the PostScript interpreter hosted by the printer.

[3] Hereafter, by "remote evaluation" we will refer to our design paradigm. Although it has been inspired by work on the REV system [8], they have to be kept definitely distinct. Our REV is a design paradigm, while the REV system is a *technology* that may be used to actually implement an application designed using the REV paradigm.

Paradigm	Before		After	
	S_A	S_B	S_A	S_B
Client-Server	A	know-how resources B	A	know-how resources **B**
Remote Evaluation	know-how A	resources B	A	*know-how* resources **B**
Code on Demand	resources A	know-how B	resources *know-how* **A**	B
Mobile Agent	know-how A	resources	—	*know-how* resources ***A***

Table 1: Mobile code paradigms. This table shows the location of the components before and after the service execution. For each paradigm, the computational component in bold face is the one that executes the code. Components in italics are those that have been moved.

In the REV paradigm, a component A has the know-how necessary to perform the service but it lacks the required resources, which happen to be located at a remote site S_B. Consequently, A sends the service know-how to a computational component B (which we call "executor") located at the remote site that, in turn, executes the code using the resources available there. An additional interaction delivers the results back.

Given the above definition, it may be argued that REV is nothing more than a special case of the client-server paradigm in which the server exports an `execute_code` service that takes a code fragment as parameter. To some extent, this is true. Yet, we believe that it is useful to distinguish between the two paradigms. In particular, it is the ability of the server/executor to offer customizable services that makes the difference. A server in the client-server paradigm exports a set of fixed functionalities. In turn, an executor in the remote evaluation paradigm offers a service that is programmable with a computationally complete language.

Code on Demand (COD)

> *Louise wants to prepare a chocolate cake. She has at home both the required ingredients and an oven, but she lacks the proper recipe. However, Louise knows that her friend Christine has the right recipe and she has already lent it to many friends. So, Louise phones Christine asking "Can you tell me your chocolate cake recipe?". Christine tells her the recipe and Louise prepares the chocolate cake at home.*

Many upcoming Internet applications are based on this paradigm. For example, consider a generic terminal that is able to download, link, and execute some code from the net. The terminal could get documents that come in a particular format that the terminal is unable to elaborate. The header of the document may contain a reference to the code that is needed to interpret the document so that the terminal, after downloading the data, could fetch the necessary code.

In the COD paradigm, component A is already able to access the resources it needs, which are co-located with it within S_A. However, no information about how to process such resources is available at S_A. Thus, A interacts with a component B contained in S_B by requesting the service know-how, which is in S_B as well. A second interaction takes place when B delivers the know-how to A, which can subsequently execute it.

Mobile agent (MA)

> *Louise wants to prepare a chocolate cake. She has the right recipe and ingredients, but she has not an oven at home. However, she knows that her friend Christine has an oven at her place, and that she is very happy to lend it. So, Louise prepares the chocolate dough and then goes to Christine's home, where she bakes the cake.*

As an example of an application that is conveniently modeled with mobile agents, consider a network management activity. The network manager would like to test the status of a set of network nodes and perform some corrective actions on each faulty node following the net topology (e.g., because the faults may be caused or propagated by adjacent nodes). Current mainstream protocols for network management, e.g. SNMP, are based on a pure client-server paradigm where a management station continuously polls and updates data on the network devices by means of very low-level `get/set` operations. As discussed in [2], this is likely to generate a huge network traffic in proximity of the management station, thus worsening the situation that management is supposed to solve. A mobile agent, in turn, could be composed of a diagnostic routine that travels among the faulty nodes performing all the needed `get/set` operations locally, without overloading the network.

In the MA paradigm, the service know-how is owned by A, which is initially hosted by S_A, but some of the required resources are located on S_B. Hence, A *migrates* to S_B carrying the know-how and possibly some intermediate results with itself. After it has moved to S_B, A completes the service using the resources available there.

The *mobile agent* paradigm is different from other mobile code paradigms in that the associated interactions involve the mobility of an *existing* computational component. In other words, while in REV and COD the focus is on the transfer of code between components, in the mobile agent paradigm a whole computational component, together with its state, the code it needs, and some resources required to perform the task, are moved to a remote site.

DISCUSSION

The mobile code design paradigms introduced in the previous section define a number of abstractions that provide an explicit model for the bindings between components, locations, and code and their dynamic reconfiguration. Our initial experience in applying the paradigms [2, 6] suggests that these abstractions are effective in the design of distributed applications. Furthermore, their independence of the particular language or system in which they are ultimately implemented is an additional asset.

Mobile code paradigms model explicitly the concept of location. The *site* abstraction is introduced at the architectural level in order to take into account the location of the different components. Following this approach, the type of interaction between two components is determined by *both* components' code and location. Introducing the concept of location makes it possible to model the cost of the interaction between components at the design level. In particular, an interaction between components that share the same location is considered to have a negligible cost when compared to interaction that is carried out through a communication network.

Most well-known paradigms are static with respect to code and location. Once created, components cannot change either their location or their code during their lifetime. Therefore, the types of interaction and its quality (local or remote) cannot change.

Mobile code paradigms overcome these limits by providing component mobility and remote code linking.

By changing their location, components may dynamically change the quality of interaction, reducing interaction costs. To this end, the REV and MA paradigms allow the execution of code on a remote site, encompassing local interactions with components located there. Components that are able to link code dynamically can extend the types of interaction they support. The COD paradigm enables components to retrieve code from other remote components, providing a flexible way to extend dynamically the behavior of a component.

SOME SCENARIOS FOR CODE MOBILITY

In this section, we depict informally some scenarios that are expected to gain great benefit from the exploitation of a mobile code paradigm, in order to show how they can help in building distributed applications.

Deployment and Upgrade of Distributed Applications

Code mobility can be exploited to support software deployment and maintenance. Traditional software engineering addresses the problem of minimizing the work needed to extend an application and to keep trace of the changes in a rational way, by emphasizing design for change and the provision of suitable development tools. In a distributed setting, however, the action of installing or rebuilding the application at each site still have to be performed locally by an operator.

The REV and MA paradigms could help in providing automation of the installation process. A scheme could be devised where the installation process is coded in a program that is transferred to a set of network nodes. On each node, the program could analyze the features of the local hardware/software platform and perform the correct configuration and installation steps.

Code mobility can go even further. For instance, let us suppose that a new functionality has to be added to an application; say, a new dialog box must be shown when a particular button on the user interface is pushed. In a distributed application designed with conventional techniques, the new functionality needs to be introduced by reinstalling or patching the application at each site. This process can be lengthy and, even worse, it is put in place even if the user never activates the corresponding functionality.

The COD paradigm could help in many ways. First, all changes would be centralized in the code server repository, where the latest version is kept. Moreover, changes would not be performed *proactively* by an operator on each site, rather they could be performed *reactively* by the application itself, that would request automatically the new version to the central repository, as soon as the corresponding functionality is activated. Hence, changes could be propagated in a *lazy* way, concentrating the upgrade effort only where it is really needed.

Customization of Services

Conventional distributed applications built following a client-server paradigm provide, by their nature, an *a-priori* fixed set of services accessible through a *statically* defined interface. It is often the case that this set of services, or their interfaces, are not suitable for unforeseen client needs. A common solution to this problem is to upgrade the server with new functionality, thus increasing both its complexity and its size, without increasing its flexibility.

The REV and MA paradigms could help in increasing server flexibility, yet keeping both the size and complexity of the server limited. In these paradigms, in fact, the server actually provides a unique service, the execution of remote code; as a consequence, it does not need to be changed, unless the language in which code is programmed and executed is changed. Hence, the server provides the maximum flexibility, in that it can execute *any* service requested by a client. By converse, each client is responsible for the correct specification of the service it needs, described by the code sent to the remote server.

This approach is well-known in certain fields of computer science. For example, it is just the way distributed relational databases work; the DBMS server is not responsible for providing answers to specific and pre-built queries, rather, the only service it provides is the execution of SQL code that comes from application programs or SQL shells acting on the client side.

However, the paradigm exploited by SQL systems differs in two respect with the ones we proposed. First, the MA paradigm defines migration of a computation that is *already in execution*, while SQL systems are more similar to REV, in that they migrate only application code. Second, our approach assumes implicitly that the languages used to implement the paradigms are computationally complete, that is not the case of SQL.

Support for Disconnected Operations

The nodes of a distributed system may be connected by a variety of physical links, whose differences in performance must be taken into account at the design level. In fact, the characteristics of the link may be part of the criteria used to select the most suitable design paradigm for the application at hand.

For instance, recent developments in mobile computing showed that low-bandwidth and low-reliability communication channels require new design methodologies for applications in a mobile setting. In complex networks where some regions are connected through wireless links while others are connected through ordinary links the design becomes complex, in that it must aim at avoiding as much as possible the generation of traffic over the weaker links.

The client-server paradigm has only one way to achieve this goal, that is, to raise the granularity level of the services offered by the server. This way, a single interaction between client and server is sufficient to specify a high number of lower level operations, that are performed locally on the server and do not need to pass across the physical link. Unfortunately, this solution is not always feasible. Moreover, it leads to an increase in complexity and size of the server, and to reduced flexibility.

The REV and MA paradigms could help because they allow, by their nature, to specify complex computations that can move across a network. Hence, the services that have to be executed by a server that resides in a portion of the network that is reachable only through an unreliable and slow link could be described in a program that should pass once through this link, being injected into the reliable network. There, it could execute autonomously and independently. In particular, it would not need any connection with the node that sent it, except for the transmission of the final results of its computation.

Improved Fault Tolerance

In conventional client-server applications, the state of the computation is distributed between the client and the server. A client program contains statements which are executed locally and interleaved with statements that invoke remote services on the server. The latter contain (copies of) data that belong to the environment of the client program, and will eventually return a result that will be inserted into the client's environment. This structure leads to well-known problems in presence of partial failures, because it is very difficult to determine where and how to react in order to recover in a consistent state.

The MA paradigm, and to some extent also the REV paradigm, could help in solving the problem of partial failures, in that they encapsulate all the state involving a distributed computation into a single component that can be easily traced, checkpointed, and eventually recovered *locally*, without any need for a global state knowledge.

CHOOSING THE RIGHT PARADIGM

The choice of the paradigm for the design of a distributed application is a critical decision. Much of the success of the development process may depend on it.

There is no paradigm that is better than others in absolute terms. In particular, the paradigms proposed here do not necessarily prove to be better than traditional ones. The choice of the paradigm must be performed on a case-by-case basis, according to the type of application. For instance, the best paradigm for a network management application could be different from the paradigm that is best suited for an information retrieval system.

For each case, some parameters that describe the application behavior have to be chosen, together with some criteria to evaluate the parameters values. For example, one may want to minimize the number of interactions, the CPU costs or the generated network traffic. In addition, a model of the underlying distributed system should be adopted to support reasoning about the criteria. For each paradigm considered the reasoning method should be applied in order to determine which paradigm optimizes the chosen criteria. This phase cannot take into account all the characteristics and constraints, which probably will be fully understood only after the detailed design, but it should provide hints about the most reasonable paradigm to follow in the design.

In the following, we will explain the ideas above by considering the example of an information retrieval application. In this application we assume that information is structured in clusters (information blocks) located on different network nodes. Each information block is managed by a component (running on that node) that coordinates the access to the data. The information that is relevant to our application may be scattered all over the network and it is assumed to be very small with respect to the size of the whole data base.

Often, such data mining activities are human-intensive, i.e., there is a person who browses through the net, interpreting documents and filtering out useful information. In this example, we suppose that the document browsing task can be carried out automatically, i.e., there is a component that scans the incoming documents and is able to filter text (and possibly images) to relate one piece of information to another, to save documents that are relevant to the search, and to follow some implicit or explicit link to other nodes to fetch other relevant information.

This application is similar to the usual process of finding some information on the Internet. So far, applications of this kind have been designed following a client-server paradigm, but one of the reasons why the idea of "mobile agent" has become popular is that it has been presented as a facility for data mining and Internet searching. In particular, it is often stated that if this kind of application would be designed with "agents" interacting with servers, performance could improve, particularly as far as network traffic is concerned.

However, this claim is supported usually only by intuition. In the following, we show that it actually holds for the REV paradigm only under certain conditions, and it never holds for MA. Surprisingly, the CS paradigm performs better than mobile code ones, if some conditions are met. We demonstrate this by building a simple model of the data mining application, determining which are the parameters we want to optimize (network traffic in this case) and reasoning about the best paradigm to exploit[4].

Description of the Model

We consider a network in which a node (e.g., a computer or a printer) communicates with another node by exchanging messages with a reliable protocol. Although in a real system the cost of communication depends on the distance between two nodes, we assume a uniform network, i.e., the cost of communication is independent of the particular node pair and is proportional to the amount of bytes that are transmitted. The cost of communication between two components that are located on the same node is null.

Furthermore, we do not consider all the implications of having code executed on remote hosts, i.e., the cost of CPU time is negligible and every machine is accessible from anywhere in the network without any access control and authentication procedure. These two assumptions are not realistic. However, for the moment we want to concentrate on bandwidth consumption only, thus we will not take these cost items into consideration.

The application is composed of a browser component that retrieves information represented by documents stored on several hosts and managed locally by a dedicated data manager component. The browser component can get either an entire document or just a document header (e.g., containing update time and keywords list) from one node by sending a request message to the data manager component located at that node. The browser keeps a "see also" list of nodes that is initialized with at least one element. At each step, the browser extracts the first node from the list and queries the data manager at that node in order to get relevant documents and references to other interesting nodes that are then merged into the "see also" list. The browser continues until all the interesting nodes have been visited.

The browser searches each node by requesting the header of all the documents that are present on that node (we assume that each data manager has an "index" document that contains references to all its documents).

[4]Note that, in doing this, we ruled out the COD paradigm which, although useful for augmenting the know-how of a component, in this case does not help us in trying to optimize communication among the data mining application and the data managers.

parameter	unit	description
N	nodes	number of network nodes that contain useful information
D	documents/node	average dimension of the data base at each network node
i	number	density of relevant information: *relevant/total* documents ratio (constant for every node).
h	bits/document	average dimension of a document header
b	bits/document	average dimension of a document body
r	bits/document	dimension of the request (includes message headers and all the auxiliary data of the request/reply).

Table 2: Parameters for the model of a simple data mining application.

It then selects only those documents that are relevant to the search and requests their bodies. The "see also" list is finally produced by extracting links from the bodies of relevant documents.

For simplicity, we make the following additional assumptions:

- all requests have a fixed length (r),
- each node holds the same number D of documents,
- the relevant information is uniformly distributed among a set of N nodes, being i the ratio between relevant and total documents,
- documents have constant length. h and b are the size of the header and the body, respectively.

The application is characterized by the parameters shown in Table 2.

Evaluating the paradigms

Client-server

If we design the application using the client-server paradigm, we will have a browser component on one node that will interact remotely with the N data managers. For each node, the browser issues D requests for document headers and $i \times D$ requests for document bodies. Thus, the total traffic is:

$$T_{CS} = ((D + iD)r + Dh + iDb)N.$$

Remote evaluation

Using the REV paradigm, the browser can obtain the execution of the filtering task on the same node that holds the data. Thus, for each node H, the browser sends a request to an executor component on H containing the code of the filter. C_{REV} is the size of this code. The executor component performs the requests to the data manager and sends the relevant document bodies across the network back to the browser. The browser extracts the "see also" list and then focuses on another node.

Therefore, for each node H there is a single request containing C_{REV} which returns only the required information, expressed by iDb. The total amount of data that move through the network is:

$$T_{REV} = (r + C_{REV} + iDb)N.$$

Mobile agent

In the MA paradigm, the browser migrates on each interesting node, performs all the interactions with the data manager and the filtering locally, and saves in its state all the relevant information and the "see also" list.

At each hop, the mobile agent carries its code and state across the network. For each hop j, the traffic is $T^j_{MA} = r + C_{MA} + S^j$, where r is the dimension of the request, C_{MA} is the dimension of the code of the agent, and S^j denotes the size of the state of the agent at hop j. More precisely, $S^j = d_{SAlist} + s + \sum_1^j iDb$, where d_{SAlist} is the size of the "see also" list, s denotes the size of other internal data structures representing the state of the computation, and the last term is the useful information collected by the agent at each visited node. Assuming that i, D, b, d_{SAlist}, and s do not depend on the node and defining for simplicity $\bar{s} = d_{SAlist} + s$, the overall traffic generated by the mobile agent is:

$$T_{MA} = \sum_{j=0}^{N} (r + C_{MA} + \bar{s} + \sum_1^j iDb),$$

that is,

$$T_{MA} = (r + C_{MA} + \bar{s} + \tfrac{N}{2} iDb)(N + 1).$$

Traffic overhead

If we isolate the amount of relevant data that must necessarily go through the network, we can reason about the traffic overhead produced by each paradigm. Thus, being $I = iDbN$ the size of the interesting information, the three paradigms produce the following overhead:

$$
\begin{aligned}
O_{CS} &= T_{CS} - I = (r + ir + h)DN, \\
O_{REV} &= T_{REV} - I = (r + C_{REV})N, \\
O_{MA} &= T_{MA} - I \\
&= (r + C_{MA} + \bar{s})(N+1) + (\frac{I}{2})(N+1) - I \\
&= (r + C_{MA} + \bar{s})(N+1) + \frac{I}{2}(N-1).
\end{aligned}
$$

Comments
It is clear that the overhead caused by the three paradigms is always proportional to the number of visited nodes. It is also clear that REV is always more convenient than MA because (1) C_{REV} is always smaller than C_{MA} because the mobile agent must also include the code that manages the "see also" list and the jump choices, (2) O_{MA} includes some terms that depend on the state of the computation. In our application the state grows with the number of hops N, thus, the overhead grows with N^2.

It is important to notice the difference between CS and REV. This evaluation should suggest the condition that makes one paradigm more convenient in terms of bandwidth consumption. The choice is expressed by the following comparison: $(r + ir + h)D \sim (r + C_{REV})$, i.e., assuming $r \ll C_{REV}$, REV is convenient when:

$$(r + ir + h)D > C_{REV}.$$

What makes the difference is that the overhead caused by CS depends on the size of the database. In particular, it is proportional to the amount of data that is necessary to perform the search (in a system with no abstracts or indexes this equals the whole data base). Instead, the overhead given by REV is bound to the size of the code, i.e., the "know-how" sent by the application. Thus, REV scales up very well for huge databases and does even better when the information is concentrated in a few clusters, i.e., N is small and D is big.

Changing the model or the application parameters
The model we analyzed is very simple. It assumes that the network is uniform and that each node is willing and able to execute code coming from foreign hosts. It also assumes that the goal is to minimize network load and no other performance metrics, e.g., response time, or other costs, e.g., implementation and deployment, are taken into account.

As a future work, this model could be extended in the following directions:

- *non-uniform networks*: we may assign different costs to different links in the network. This way, we may simulate disconnected/mobile computing, or networks characterized by clusters of nodes with high-speed connections among nodes within a cluster, and unstable and slow links among clusters.
- *security/CPU costs*: we can assign a non-null cost to the execution of code on remote hosts. This cost should include a fixed part that accounts for security procedures, e.g., authentication or code analysis, and a variable part proportional to the CPU usage.
- *memory usage*: we may assign a cost proportional to the amount of memory used by the executor of the code.

CONCLUSIONS AND FUTURE WORK

On the basis of our ongoing research on MCLs, in this paper we described some design paradigms that can be derived by abstracting from the details of such languages. More important, we believe that the paradigms we described are independent of the technology used to implement them, and that, to some extent, they can be valuable even when specific mobile technology is not used at all. In addition, we believe that there is no "best" solution to the problem of designing an application with a mobile code paradigm. Rather, the best solution has to be found on the basis of a careful evaluation of the application requirement against the peculiar features of each paradigm. We gave only some hints about the process of selecting the right paradigm for a given application, and surely there is room for further work to be done in this field.

We plan to extend the work described in this paper in the following directions:

- There is a strong need for real-world applications developed using mobile code technology and/or design paradigms, that should be used as a testbed to verify the advantages of the "mobile" approach. For this purpose, we plan to implement a mobile code application for network management, along the lines described in [2].
- We will extend the repertoire of design paradigms described in this paper and provide assessment criteria that will allow each paradigm to be evaluated with respect to the quality criteria of a given application.
- It is an open question whether the paradigms described here, which are inspired by the feature provided by MCLs, cover all the conceivable design paradigms for code mobility. Thus, we are currently interested in investigating the minimal abstractions needed to express all the possible interaction patterns involving code mobility at design

level, and their corresponding mapping at the language level.

ACKNOWLEDGMENTS AND DISCLAIMER

This work could not have been made without enlightening discussions with several people, including G. Cugola, A. Fuggetta, and C. Ghezzi. Another *vital* contribution comes from the "cake makers", Cristina and Luisa, who in real life are Picco's fiancee and Vigna's wife, respectively. While giving a talk about this work, one of us was informed that our examples seem to convey an image of the woman which is not "politically correct". It is not a matter of political correctness. Simply, we thought that nobody—at least among our colleagues—would have believed in examples in which we were in the role of cooks. Furthermore, we found that using the names of our significant others (who actually bake delicious cakes) if compared with sentences like "X wants to prepare a cake", was far better than dealing with aseptic symbolic names.

REFERENCES

[1] G. Abowd, R. Allen, and D. Garlan. Using Style to Understand Descriptions of Software Architecture. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, December 1993.

[2] M. Baldi, S. Gai, and G. P. Picco. Exploiting Code Mobility in Decentralized and Flexible Network Management. In *Proceedings of the First International Workshop on Mobile Agents (MA97), Berlin, Germany*, April 1997. To appear.

[3] A. Black, N. Hutchinson, E. Jul, and H. Levy. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[4] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems*. Lecture Notes on Computer Science, 1997. To appear.

[5] B. T. et al. Facile Antigua Release Programming Guide. Technical Report ECRC-93-20, European Computer-Industry Research Centre, Munich, Germany, December 1993.

[6] C. Ghezzi and G. Vigna. Mobile Code Paradigms and Technologies: A Case Study. In *Proceedings of the First International Workshop on Mobile Agents (MA97), Berlin, Germany*, April 1997. To appear.

[7] R. S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM'95 Workshop on Intelligent Information Agents*.

[8] J. W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[9] D. Johansen, R. van Renesse, and F. Schneider. An Introduction to the TACOMA Distributed System - Version 1.0. Technical Report 95-23, "University of Tromsø and Cornell University", June 1995.

[10] J. Kramer. Distributed Software Engineering. In *Proceedings of the 16th International Conference on Software Engineering*, Sorrento (Italy), May 1994.

[11] G. Magic. *Telescript Language Reference*. General Magic, October 1995.

[12] B. Mathiske, F. Matthes, and J. Schmidt. On Migrating Threads. Technical report, Fachbereich Informatik Universität Hamburg, 1994.

[13] OMG. CORBA: Architecture and Specification, August 1995.

[14] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1995.

[15] D. Perry and A. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, October 1992.

[16] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice Hall, 1996.

[17] M. Straßer, J. Baumann, and F. Hohl. MOLE - A Java Based Mobile Agent System. In *Proceedings of the Second International Workshop on Mobile Object Systems*, Linz, July 1996.

[18] Sun Microsystems. *The Java Language Specification*, October 1995.

[19] C. F. Tschudin. *An Introduction to the* `MO` *Messenger Language*. University of Geneva, Switzerland, 1994.

[20] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report TR-94-29, Sun Microsystems Laboratories, November 1994.