Is Code Still Moving Around? Looking Back at a Decade of Code Mobility

Antonio Carzaniga
Univ. of Lugano, Switzerland
antonio.carzaniga@unisi.ch

Gian Pietro Picco
Univ. of Trento, Italy
picco@dit.unitn.it

Giovanni Vigna
UC Santa Barbara, USA
vigna@cs.uscb.edu

Abstract

In the mid-nineties, mobile code was on the rise and, in particular, there was a growing interest in autonomously moving code components, called mobile agents*. In 1997, we published a paper that introduced the concept of mobile code paradigms, which are design patterns that involve code mobility. The paradigms highlighted the locations of code, resources, and execution as first-class abstractions. This characterization proved useful to frame mobile code designs and technologies, and also as a basis for a quantitative analysis of applications built with them. Ten years later, things have changed considerably. In this paper we present our view of how mobile code evolved and discuss which paradigms succeeded or failed in supporting effectively distributed applications.*

1 Introduction

In 1997, midway through our doctorate, we presented a paper at the 19th International Conference on Software Engineering (ICSE) about "Designing Distributed Applications with Mobile Code Paradigms" [3]. The research reported there and later included in a more comprehensive paper [6] became, for some of us, an integral part of our doctoral dissertations [12, 19].

Ten years later, our paper was named the *Most Influential Paper from ICSE'97*. We were obviously very pleased with the news—a little less with the implicit realization that ten years had passed. In this decade many things have changed. This paper presents our thoughts on what happened to mobile code paradigms and our view on the current state of the art.

Mobile code and mobile agents were extremely "hot" in the mid '90s, when we wrote our paper. Novel applications and technologies were appearing at a high rate, each time somehow shifting the common-sense notion of what it meant to dynamically move the code (and/or the state) of a program. Indeed, the main contribution of our ICSE'97 paper was precisely to abstract away from the details of technologies and applications, and identify some recurring *design paradigms* featuring code mobility.

This paper is devoted to the analysis of what happened to code mobility in the last decade. First, in Section 2, we set the scene for the rest of the paper by defining *our* notion of code mobility and related design paradigms. Then, for each paradigm, in sections 3 through 5 we compare the expectations of the research community a decade ago against today's reality. In Section 6, we try to understand *why* things developed the way they did, and in particular we discuss the causes that led to the rise (or fall) of each paradigm. We end the paper in Section 7 with some concluding remarks.

Finally, the reader should be warned that this paper is less of a comprehensive and impartial report, and more of a collection of personal *opinions*, sometimes supported by anecdotal facts and direct experience.

2 Code Mobility in a Nutshell

In our ICSE'97 paper we defined code mobility as

> the capability to reconfigure dynamically, at runtime, the binding between the software components of the application and their physical location within a computer network [3].

This entails that executable content is moved across the network in order to execute (part of) the functionality of an application. The idea behind mobile code is that by bringing the code close to the resources needed for a certain task it is possible to perform the task in a more effective way.

2.1 Mobile Code Technology

Although in principle code mobility could be implemented in an *ad hoc* fashion using standard facilities, dedicated technologies provide the programmer with direct means to transfer code (and possibly state) and automatically execute it. Several technologies were proposed at the time, including new programming languages, extensions to existing ones, and operating system libraries.

Mobile code technologies can be analyzed by considering the ability to transfer the state of an execution thread (or *execution unit*, to use a more general term) as the criteria to discriminate among them, as we did with another paper [4, 6]. Therefore, we say that a technology supports *strong mobility* if it allows executing units to move their code and execution state (e.g., the stack and instruction pointer of a thread) to a different site. When an executing unit must be transferred to a remote site, it is suspended, transmitted to the destination site, and resumed there. A technology supports *weak mobility* if it allows an executing unit in a site to be bound dynamically to code coming from a different site (i.e., the code can be moved and executed automatically), but no execution state is transferred across the network. This means that even though some data state could be transferred, the executing unit would need to be restarted upon arrival.

2.2 Mobile Code Design Paradigms

The content of our ICSE'97 paper was inspired by the observation that it is possible to abstract away from the detail of each technology and reason about "common ways," or *design paradigms*, for structuring applications involving code mobility.

The design paradigms we defined are based on architectural abstractions such as *resource components* (both *data components* and *code components*, the latter representing the "know-how"), *computational components* (threads of computation), *interactions* (events that involve two or more components), and *sites*, (execution environments representing a "location"). For a more detailed definition of these components, please refer to the original paper [3].

These abstractions can then be used to describe different ways of structuring a computation. More precisely, the paradigms focus on 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, involved in the delivery of the service. 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. Key to the discussion is the fact that, to accomplish the service, these elements must be present at one site *at the same time*.

The classic *Client-Server* paradigm can be described in these terms. Here, a computational component B (the server) offering a set of services is placed at site S_B. The resources and know-how needed for service execution are hosted by 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 reaction, 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 result that is delivered back to the client with an additional interaction.

Mobile code paradigms make different use of interactions and service constituents. In the *Remote Evaluation* paradigm, a component A owns the know-how necessary to perform the service but 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 located at the remote site. This component, in turn, executes the code using the resources available there. An additional interaction delivers the results back.

In the case of the *Code On Demand* paradigm, component A is already able to access the resources it needs, which are co-located with it within S_A. However, no knowledge about how to process such resources is available at S_A. Thus, A interacts with a component B located at 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.

Finally, in the case of the *Mobile Agent* paradigm, the service know-how is owned by A, initially hosted by S_A, but some of the required resources are located on S_B. Therefore, the computational component 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.

Table 1, reproduced from the original paper, shows how the characterizing elements of the various paradigms are relocated as a result of the paradigms' interactions.

These design paradigms, along with the distinction between strong mobility and weak mobility, were widely accepted. This is partly due to the fact that they provided terminological grounds in a period where buzzwords, hype, and confusion were the norm in the field. Ten years later, the dust has settled and most of the hype is gone. In the following sections, we analyze what was the perception at the time about strengths and weaknesses of each paradigm, along with their actual impact on the design of real-world applications.

3 Mobile Agent

Among the paradigms involving code mobility we considered in our paper, the mobile agent one is arguably the one that attracted the most interest. Indeed, the idea of an application component autonomously wandering through the net tickled more than one mind with the way it radically changes the mainstream criteria of application design.

At the time when we wrote our ICSE'97 paper, mobile agents were all across the board. Mobile agent systems were mushrooming, somewhat inspired by the concepts made popular by Telescript [22], and later enabled by

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 design paradigms [3]. This table shows the location of the components before and after 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.

the fundamental building blocks provided by the Java platform. Novel applications were envisioned, formal theories of mobile agents rapidly became trendy, and conferences devoted to the topic started to appear.

A decade later, the future of the mobile agent paradigm looks definitely less bright. A clear symptom is the shrinking of the research community involved. The premiere conference on the topic was the IEEE International Conference on Mobile Agents (MA), whose last edition was held in 2002 after the steering committee (involving some of the authors of this paper) observed that the number of submissions, but more importantly their quality and originality were significantly declining. Interestingly, most of the key contributors to the field are today involved in other research areas, often with little or no relation to mobile agents.

Hereinafter, we analyze how mobile agents have evolved in the past decade by focusing on the two most prominent dimensions, i.e., applications and technology.

3.1 Mobile Agent Applications

Mobile agents were proposed as the enabling paradigm for a plethora of applications, where the paradigm's characteristics were supposed to bring key advantages—according to some researchers, to the extent of potentially revolutionizing the development of distributed applications. Those were times when, shortly after the explosion of the World Wide Web, many sought a "killer application" for mobile agents in the hope to get a similar breakthrough. Ten years later, it is safe to say that a killer application for mobile agents has not been found (yet?).

One can surely argue pragmatically—and some did—that mobile agents are indeed useless because there is not a killer application for them. It turns out, however, that the quest for a killer application that animated the rise of the field (and periodically resurfaced during its fall) is somewhat sterile, and, ultimately, based on an ill-defined issue. Indeed, the notion of "killer application" for mobile agents was interpreted by many as one that *cannot be built without mobile agents*. However, this idea was explicitly rejected since the early developments of this research area. In fact, Harrison *et al.* argue in their seminal paper that "there is nothing that can be done with mobile agents that cannot also be done with other means." [10] We essentially agreed with this view, and we continued along the same line, asserting that mobile agents (and in general architectural paradigms for code mobility) should be considered as simply another design tool available to the application developer. Therefore, mobile agents are not *enabling* new functionalities *per se*, but rather they may lead to faster, more flexible, or more efficient *implementations* of the same functionality.

Side-by-side with the fanatics of the "killer application" concept, this view somehow percolated through the research community, which started to look for application domains where one could leverage the benefits put forth by mobile agents. Quickly, some applications caught on as common informal benchmarks, and were used in discussions among researchers to highlight advantages or argue over design tradeoffs.

Two such common applications were information retrieval and e-commerce. In both cases, the idea is that a mobile agent is first assigned a query, for example, to find a specified document or, in the case of e-commerce, to find the Web site that sells a given music record at the best price. Then, the mobile agent is launched into the network where it somehow finds its way towards relevant sites, queries them locally, and eventually returns home having collected the requested information.

The general advantages of the mobile agent approach in

these applications were claimed to be:

1. The ability to reduce the communication overhead, by replacing remote interactions with local ones.

2. The greater flexibility of the approach, enabling on-the-fly customization of the server side with client code.

3. The autonomy of mobile agents, which were assumed to be able to keep the user entirely out of the loop, by determining independently the next hop based on the intermediate results.

Unfortunately, in practice we can observe that:

1. Depending on the amount of data to be returned, the communication overhead can actually increase, since the mobile agent state grows at each hop, as we showed in our ICSE'97 paper and in a follow-up [2].

2. On-the-fly customization seldom requires the relocation of a running component with its execution state. Therefore, it is achieved more easily and efficiently by relocating only code, leveraging either the code on demand or remote evaluation paradigm. Indeed, as discussed later, these paradigms have been quite successful at this.

3. In practice, in the aforementioned applications it is quite difficult to come up with sequences of queries where the result of one determines the input of the other. More precisely, it is difficult to identify situations where this ability is necessary or even advantageous. And, finally, even when this is the case it is quite difficult to encode the corresponding behavior without involving the user in the intermediate steps.

Also, technological issues hampered these two applications. Both suffered from the lack of widespread mobile agent platforms integrated in Internet technology (e.g., browsers). In addition, e-commerce implies security, and, unfortunately, devising solid security mechanisms is very hard in the presence of mobile agents, as it will be discussed later in this section. As for information retrieval applications, the widespread acceptance of web-based search engines (most notably, Google) made search engines based on mobile agents not worth the effort.

Interestingly, in some cases the notion of a mobile agent able to autonomously roam the network has been replaced by weaker notions. For instance, the research field of *active networks* initially explored the notion of *capsule*, which is a network packet that is able to self-route through the network, by carrying the routing logic along with the packet data—effectively turning a network packet into a mobile agent [17]. Nevertheless, it was soon recognized that this strategy was too inefficient in general. An alternative design appeared shortly after, where the packet carries a tag referring to the code containing the routing logic [21]. This code is *not* carried along with the packet, rather it can be dynamically downloaded and linked, as well as cached and reused in the case of functionally-related packet streams. In terms of the mobile code paradigms we defined, this new proposal replaces the mobile agent paradigm embodied by the capsule with a solution based on the code on demand paradigm.

So, are mobile agents just plain useless? Not really. The advantages brought by the paradigm are real. For instance, a follow-up of our ICSE'97 paper examined the benefits of mobile code paradigms in the domain of network management [2]. In this case, mobile agents hold the potential for huge communication savings, thanks to their ability to perform *semantic compression* and to their *decentralized* mode of operation.

Semantic compression is a term used to express that data are compressed based on their content, unlike common compression algorithm that instead work by considering only bit patterns. In the context of network management, for instance, a (common) query that requests the highest loaded network interface in the network would require a mobile agent to carry only a single scalar value as part of this state. Therefore, it does not incur the increasing cost we mentioned for the information retrieval application.

Moreover, protocols for network management are typically designed using a client-server paradigm, and therefore involve the network management station in each interaction. This can easily cause congestion when a network problem arises. Mobile agents help greatly in reducing the communication overhead on the management station, since the computation performed by the mobile agent is performed away from it.

Therefore, there *are* cases where mobile agents can be useful. In these cases, they often bring significant advantages. Unfortunately, these cases are rare, in relative terms, and none of them is observable today in the context of a widespread application. Finally, even when there are advantages, the presence of established technology and commercial interest (as in network management), or the relatively poor state of the art in mobile agent technology hampered adoption. The latter aspect is discussed next.

3.2 Mobile Agent Technology

Interestingly, the lack of applications was and still is complemented by a high number of platforms for developing mobile agents. In 2000, the census in the Mobile Agent List[1], maintained by Fritz Hohl at the University of Stüttgart, contained 72 systems. This number is even more

[1] `http://www.reinsburgstrasse.de/mal/mal.html`

stunning if we consider that the term "mobile agent" started to emerge in 1994, when Telescript [22], the first mobile agent system, and Java, the preferred development language of later systems, came out. Therefore, an average of 12 mobile agent system per year were developed between 1994 and 2000—basically one per month!

As the reader can imagine, among these systems, many of which are no longer supported or even available, very few advanced the state of the art. The reason for this "explosion" (or pollution) of mobile agent systems can be understood by observing that most of them share the following key characteristics:

1. They are written in Java.

2. They support *only* the mobile agent paradigm and provide a limited set of features, mostly restricted to the ability to relocate a component through a `go` method.

In reality, it is *very* simple to put together a mobile agent system in Java. This language was designed with distributed systems in mind, and already provides most of the necessary building blocks, including multi-threading for the execution of the agents, serialization and networking for the transmission of the agents, and—most important—programmable class loading, which enables the dynamic linking of foreign code and, as such, is the cornerstone of mobile agents and code mobility at large.

However, Java had also significant drawbacks. To begin with, Java supports only weak mobility, while a natural match for the mobile agent paradigm is an implementation technology supporting strong mobility [7]. Researchers developing mobile agent systems came up with different ways of circumventing the problem:

1. Modify the Java virtual machine so that it provides strong mobility. A few implementations exist [1], but with Sun adamantly refusing to introduce strong mobility in the Java platform, many of the advantages of a portable system disappear.

2. Implement some form of preprocessing or code rewriting to instrument the original application code with the ability to automatically save its state in application variables and correctly resume the execution flow. Some neat solutions exist [13], but the problem is that they severely bloat the resulting code. When mobile agents are used to reduce communication overhead, this becomes a serious disadvantage.

3. Claim that strong mobility is not really necessary. This is clearly the easiest solution for the developer of the mobile agent system, but puts the entire burden on the *application* developer. Without appropriate support, the benefits brought by the expressive power of the mobile agent abstraction are somewhat hampered by the extra work required to the programmer.

In other words, the availability of ready-to-use building blocks had a rather perverse effect on the community. Instead of simplifying development, and freeing resources to be invested in more innovative aspects of mobile agents, the research field handcuffed itself with Java (with a few notable exceptions such as D'Agents [9], formerly known as Agent Tcl), and therefore essentially did not advance significantly the state of the art.

To grasp in more detail the extent of this phenomenon, it is worth comparing briefly the features of Telescript, the system that first introduced the concept of mobile agents, and Aglets [11], probably the most popular among Java-based systems. Table 2 concisely compares the two systems, which appeared almost four years apart from each other. The Telescript language and associated run-time were developed from scratch with the precise goal of supporting the mobile agent abstraction, and therefore included features that are key in enabling it, and that are very difficult to provide on top of existing platforms. As a result, Telescript was ahead of its time, and is still more expressive and sophisticated than existing mobile agent systems.

By and large, all of these observations hold, with few exceptions, for the vast majority of existing mobile agents systems. Surprisingly enough, however, people still build Java-based mobile agent systems that essentially repeat the same technology story already told ten years ago, or tackle problems (e.g., locating mobile agents, or making them interoperate—a weird problem for a field where there is not even *one* widely-deployed system) that are largely inessential in boosting a wider adoption of the paradigm.

Although the points above illustrate several opportunities for supporting mobile agents that have been missed by the field, from a practical standpoint one of the major causes commonly associated with the lack of a widespread adoption of mobile agents is security [18]. Mobile agents exacerbate existing security issues and, in addition, introduce completely new ones.

One issue is that, to enforce access control, a mobile agent must be authenticated with respect to some identity. The problem is that there are many identities associated with a mobile agent. For example, an agent may be associated with the agent developer, the agent's code signer, the agent dispatcher, and the host the agent visited last. It is not clear which identities should be authenticated and how the access control mechanisms should take into account this information. Even if a suitable general model is devised, the complexity of such a model would make the access control configuration process extremely error-prone.

The most difficult (and novel) problem, however, is that mobile agents traveling across multiple hosts to complete

Telescript (1994)	Aglets (1998-2004)
Strong mobility	Weak mobility
Resource control is built into the run-time	No resource control at the run-time level
Ownership rules determine migration of bound objects and security	Security and object bindings explicitly dealt with by the developer
Instance-level member protection separates instances of the same agent class	Only conventional class-level member protection is allowed

Table 2. "Old" vs. "mainstream" mobile agent technology.

their tasks are vulnerable to a number of attacks coming from malicious executing environments. For example, a malicious host can modify the code or memory image of an agent to change the way the agent behaves. The result of this "brainwashing" attack would be the creation of a malicious agent whose actions are attributed to one of the identities initially associated with the agent. This type of attack is extremely difficult (if not impossible) to detect and prevent [14]. Therefore, security is no exception with respect to the current state of mobile agent technology, and the available solutions still fall short of expectations.

In conclusion, the sad reality is that, after a decade of mobile agent research and many systems, a reliable, expressive, and secure mobile agent system is still yet to come. Whether this is the cause or the effect of the lack of mobile agent applications is difficult to ascertain. However, the net effect is that the mobile agent paradigm, albeit being the most powerful and intellectually stimulating, still has virtually no real-world application to date, and therefore bears only a very limited impact on common practice.

4 Code On Demand

Code On Demand is a mobile code paradigm in which the code for the execution of a task is requested by the client and provided by a (code) server. When the code is received by the client, it is executed in the client's context. The code on demand paradigm is similar to the remote evaluation paradigm in the sense that no execution state is exchanged between client and server, and therefore the execution remains confined to a single site (the client in code on demand, the server in remote evaluation). Despite the similarities, the code on demand paradigm enjoyed a much greater success, and is arguably, by far, the most widely-used mobile code paradigm.

The seeds of the success of the code on demand paradigm in the application domain were already apparent by 1997. These are discussed next, followed by an analysis of the advantages of this paradigm.

4.1 Code On Demand Technologies and Applications

Since the early 1990s, but more prominently in the second half of the nineties, a number of mainstream technologies that directly supported the code on demand paradigm started to emerge. The Java language environment, with its dynamic class-loading feature, was perhaps the most representative one. Java's primary design goal was to realize an architecture-neutral platform for networked applications. The following are some key elements of the rationale behind the development of Java as explained by its main designers:

> The massive growth of the Internet and the World-Wide Web leads us to a completely new way of looking at development and distribution of software. [...] New code modules can be linked in on demand from a variety of sources, even from sources across a network. [...] Interactive executable code can be loaded from anywhere, which enables transparent updating of applications. The result is on-line services that constantly evolve [and] remain innovative and fresh [8].

The idea was therefore to create an application-deployment environment for *dynamic applications*. That is, applications extended at run-time through dynamically-linked modules, or even applications assembled entirely at run-time. Notice that the code on demand paradigm was a central element of the design of the Java platform, with the specific intent to reduce deployment and management costs for feature-rich applications.

The Java language environment was also soon incorporated in Web browsers, extending the idea of dynamic applications to the Web. The use of Java technology in Web browser was a strategic move in the right direction on the evolution path of the Web. It was recognized that the Web would evolve from a collection of interlinked but static and almost exclusively textual documents, to a dynamic and interactive platform for the delivery of multi-media content as well as applications. The natural implication of the vision of a rich and interactive Web was that Web browsers had to become active components, and Java was one of the first steps in that direction.

Despite being so forward-looking as a Web technology, Java was not as successful on the Web as some initially predicted. Instead, the technology of client-side scripting languages, whose most common representative is JavaScript, was to become the chief motor behind active Web content and code on demand. JavaScript, which despite the common prefix has very little in common with Java, is a scripting language whose execution environment is tightly

bound to HTML documents and their rendering within a browser. In fact, JavaScript was designed specifically to support active and interactive Web documents, rather than as a general-purpose language.

Our 1997 paper did not predict the success of active Web content. In fact, the example scenarios we envisioned for the code on demand paradigm were consistent with the Java vision of general-purpose dynamically extensible applications. Ten years later, however, active and interactive Web content has become the predominant realization of the code on demand paradigm.

Nevertheless, the idea of dynamically loaded *applications* is all but extinct. In fact, these applications, which have since been renamed *rich Internet applications* or "RIAs," seem ready for commercial success, and therefore are receiving great attention even in the mainstream popular press [5]. Concrete examples are some of the network-based applications offered by Google, from e-mail readers and calendars to word processors and spreadsheet applications. All these applications make extensive use of the code on demand paradigm. Most of them are currently implemented through JavaScript, although the technology of Web client-side scripting is evolving very rapidly, too. Not surprisingly, the evolution is in the direction of a more general platform, like Java. Examples of this evolution are new languages and environments for Web browsers, such as Adobe's ActionScript language for the Flash player.

In conclusion, it is interesting to note that the code on demand paradigm captures the essence of the success of both dynamic Web content and dynamic applications, regardless of the success of specific technologies.

4.2 Advantages of Code on Demand

Given the success of code on demand, it is natural to ask what are the advantages of this paradigm, independently of its applications.

There are two reasons for its success. The first reason lies in the fact that the actual implementation of this paradigm does not require significant engineering difficulties. From a mere execution standpoint, the essence of the code on demand paradigm can be supported by an interpreter equipped with a networked dynamic loader. This technology is well-understood, and today well-supported by mainstream platforms (e.g., Java and .NET). From a security standpoint—the crux of mobile code approaches—some precautions must be taken. At a minimum, the execution environment of the client should be able to authenticate the code coming from the server. A more serious security concern is the ability of the client to limit the behavior of code executed on demand. However, once more, these are well-understood problem where good and readily-available technological solutions exist (e.g., certificate-based schemes, as well as virtualization and sandboxing).

The second reason for the success of the code on demand paradigm is that it intrinsically fosters good load balancing properties. Indeed, client-server places the burden for computation entirely on the server, which holds the resources and know-how necessary to the service. Instead, the code on demand paradigm helps in moving the computation as much as possible on the client. Under the assumption that there are more clients than servers, code on demand does a better job at distributing the computational load away from the server, therefore greatly improving its scalability. Interestingly, this is an advantage that code on demand enjoys not only over client-server, but also over remote evaluation. To some extent, this explains why code on demand has been much more successful than remote evaluation, discussed next.

5 Remote Evaluation

In the remote evaluation paradigm, the component who initiates the computation, and is interested in the results, sends code to a remote executor component that has access to the resources needed for the computation. Remote evaluation can also be seen as a client-server model where the service being provided is the execution of code—in a way, a meta-service. Remote evaluation has the advantage of being simpler than the mobile agent paradigm (which requires code and execution state to be transferred and managed), while, at the same time, it provides the flexibility that allows for the creation of on custom services and the execution of complex tasks germane to code on demand.

In addition, remote evaluation shares with mobile agents the ability to perform semantic compression, which we mentioned already in Section 3. In the interactions between two components that involve an exchange of data, this can be "compressed" by transferring the executable code necessary to produce or filter the data. This advantage is to some extent the *raison d'être* of remote evaluation, and indeed was one of the key ideas behind the system that first proposed the concept [16]. The most obvious example is the execution of a query on a database server: in almost every practical case, the size of a database far exceeds the size of a query plus the size of its output. The same argument applies to the execution of PostScript code on a printer: the client only needs to ship a short, device-independent representation of a document to a printer, which is then used to generate a typically much larger amount of data in the form of the raw bitmaps processed by the printer device.

As of today, however, the landscape of technologies and applications that use the remote evaluation paradigm is largely unchanged. In our 1997 paper we mentioned three examples: remote shell, database queries, and PostScript printing. All these examples are basically still observable

today, along with some more modern applications. For example, remote evaluation is used in a number of measurement and monitoring applications such as ScriptRoute [15]. Also, the PlanetLab network[2] can be seen as a large platform for applications based on remote evaluation. Other growing application domains where remote evaluation is used extensively as a design paradigm are so-called "commodity computing" and "grid computing." In these application domains, tasks that logically belong to a single (client) application are transferred to remote servers for execution.

Therefore, remote evaluation essentially bears the same impact on common practice as it did a decade ago. Interestingly, the vision of a general-purpose programming support for remote evaluation put forth originally by Stamos and Gifford [16], which was a sort of RPC where the code is itself a parameter, did not really materialize. In all the application domains above, in fact, the paradigm is embedded in an *ad hoc* fashion into applications, rather than being supported through a general application-programming interface.

6 Discussion

So, is code still moving around? Yes, but not quite in the way many expected ten years ago.

Regardless of their individual success, mobile code paradigms *collectively* had an impact on academia and research, feeding new ideas and new ways to look at problems. In addition, mobile code techniques are used by various research communities in a number of research projects, for achieving rather disparate goals. However, if we look pragmatically at how much mobile code paradigms can be "observed in action" in today's applications, reality tells a different story from the expectations of a decade ago.

At that time, some of the most enthusiastic supporters of mobile code were certain that the "old" client-server paradigm would have soon died to give way to new models of distributed applications, and that the more sexy and appealing mobile agent paradigm would have revolutionized distributed computing. Of course, that did not happen. Instead, client-server—the paradigm without code mobility—is very much alive, and thanks to new interface definition languages and transport mechanisms, it was re-invented in the form of so-called Web services. At the other extreme, mobile agents are essentially relegated to a niche in academic research with essentially no practical applications.

The story is different for mobile code paradigms that move only code, that is, the paradigms naturally supported by weak mobility. Remote evaluation is in use for basically the same tasks as a decade ago. Code on demand, which already had its share of popularity thanks to Java-enabled Web browsers, has been extremely successful as a basis for active Web content. Besides its use as the language of active Web pages, code on demand was expected to catch on and percolate into mainstream computing, leading to a revolution in the business of desktop office applications. Thin clients were envisioned as the main computing platform, with network access to highly componentized versions of common productivity applications whose bits and pieces could be separately bought and downloaded on-demand. This vision did not become a reality. However, things are once again moving in that direction.

Another way to measure indirectly the success of the various paradigms is to look at the technology supporting them today, along the reasoning that a new paradigm, if successful, triggers a change in technology to better support it. This change is evident for the client-server paradigm: remote procedure call (RPC) was developed to simplify the implementation of applications based on the client-server paradigm, and in turn boosted its acceptance. Of the mobile code paradigms we considered, only code on demand gained enough strength to induce the development of dedicated technology. This happened along two lines: on one hand, domain-specific technology appeared to support the paradigm in Web environments. On the other hand, code on demand became an important feature in several commercial programming environments and middleware, notably Java/RMI and .NET, opening up its use in general-purpose applications. This did not happen for remote evaluation, which did not really evolve out of specialized application domains where the existing technologies (e.g., scripting or query languages) are usually enough to get the job done. As for mobile agents, we already discussed how the lack of real applications stifled the development of reliable technology supporting this paradigm.

The question, however, is: What are the reasons for the rise or fall of mobile code paradigms?

One argument that is often put forth is that the fate of ideas—in our case, mobile code paradigms—is eventually determined by their applications. For instance, the pragmatic stand of many people about mobile agents is simply that *if using mobile agents made sense from a functionality or performance standpoint, today we would see them in applications*. Although this is a valid argument, and one that we often bring up ourselves, ten years may simply be too short of a time span to assess the impact of an idea. The history of science is full of seemingly dead-ended ideas that were only much later successfully applied in other contexts.

From a more technical standpoint, we believe that the key to the rise or fall of the various paradigms is *complexity*, in its many forms. In particular, next we discuss the complexity of mobile code paradigms as it relates to their execution platform, application development, management, and run-time efficiency.

One complexity lies in the implementation of the lan-

[2] `www.planet-lab.org`

guage and execution platform. For instance, mobile agent platforms require dedicated support to handle the transfer of the execution state as well as its binding to local or remote resources. This feature is not available in Java precisely because it would have introduced significant complications without a motivating application in sight. In contrast, the code on demand and remote evaluation paradigms are simpler, as they only require the transfer and dynamic execution of code, which is supported essentially by any scripting language and by languages that provide class-loading facilities, as it is the case for Java. The client-server paradigm requires an even simpler environment, where the components involved need only agreement about the protocol used to invoke the service and exchange data.

As for the complexity of developing applications, mobile agents express a natural and intuitive unit of autonomous computation, which is also a natural unit of modularization for application code. However, this decomposition assumes that every agent may move from node to node independently, making decisions based on its accumulated state or on interactions with other static or mobile components whose characteristics and capabilities are *a priori* unknown. As a result, the emergent collective behavior of an application made of multiple mobile agents may be difficult to predict and hard to control. Instead, all the other paradigms leave unchanged the familiar criteria of application decomposition, independent of mobility. A task can be invoked remotely with client-server, it can be pulled from the server in code-on demand, or it can be delegated to the server using remote evaluation. These differences notwithstanding, the overall structure of a distributed application remains essentially the same as in a completely centralized application.

Moreover, there is the complexity of management. Management of mobile agents includes the initial configuration and deployment of the agents, which are complex activities for the same reason that development and analysis are complex, but also all the typical phases in the life cycle of a software system, including update and removal. For example, how would one manage a mobile agent component whose implementation has become obsolete? How can one update an agent that is executing "in the wild" at some unknown location? The complexity of these tasks becomes clear when compared to the client-server paradigm or even to paradigms that use weak mobility. The fact that computational components do not move, or that they might move at most within a single interaction to execute a single task, allows for a simple binding between components and hosts. In other words, it is statically possible to figure out what is executing where, and this, in turn, allows for simpler configuration, testing, deployment, and update of components as well as of entire applications.

Security is another major management concern, making a difference especially between client-server and mobile code paradigms. Providing support for arbitrary code execution is very difficult, and a single mistake could cause major security incidents. Not surprisingly, a Web search for "arbitrary code execution" returns only references to security vulnerabilities and no mention of systems and technologies to support code mobility and distributed applications. Arguably, security was the primary reason for the failure of mobile agents, and conversely it promoted the simpler client-server paradigm. Client-server is manageable from an authentication and authorization point of view because the server side has full control on which resources are accessed and by whom.

On the contrary, mobile code paradigms support the execution of arbitrary code and, therefore, require more complex authorization mechanisms and policies. For example, consider a service that provides a document given a unique identifier for the document. Consider now the same service that accepts an arbitrary fragment of code that can access all the public documents on the server. It is clear that the former is easier to secure than the latter, because the designer needs to anticipate only the abuses coming from the service inputs (e.g., a maliciously-formatted document identifier triggering a memory error), and can ignore many more abuses that would come from the execution of arbitrary code (e.g., code that performs a denial-of-service attack or uses the service as a trampoline to attack other systems).

Finally, the last dimension of complexity we consider is run-time complexity, and in particular communication complexity, measured as the amount of bits that an application transmits over the network. One of the main ideas behind code mobility is that it can reduce the communication complexity of an application thanks to the semantic compression obtained when moving code instead of data. Following this intuition, mobile agents were initially proposed as an ideal solution for the retrieval of information from distributed databases. However, as we showed in our ICSE'97 paper, mobile agents are less than ideal for that task, for which remote evaluation achieves a better communication complexity in most cases. It is important to notice that this result does not necessarily generalize to other applications, and that no paradigm is inherently more efficient than another. For instance, we found different tradeoffs in the field of network management [2]. Yet, the characterization of the mobile code (and client-server) paradigms can serve as a basis for the quantitative analysis of communication complexity.

7 Conclusions

In 1997, many thought that mobile code (and especially mobile agents) was going to radically change the design and implementation of distributed applications. As often happens when a potentially disruptive idea is introduced,

code mobility was accompanied by a considerable amount of hype and confusion regarding terminology, applicability of approaches, and suitability of technologies.

Our ICSE'97 paper was an attempt at identifying some well-defined abstractions that could be used to reason about mobile code designs. In addition, we showed that, by using these abstractions, one can model the behavior of mobile code applications to identify and understand the advantages and disadvantages of different designs, including those based on the traditional client-server paradigm.

Ten years later, we can see that things did not go the way the advocates of mobile code had predicted. Although mobile code is now a commonly accepted tool for many distributed applications, and despite the significant impact of mobile code on the research community, client-server is still the prevailing design paradigm. Nevertheless, the basic message of our paper seems still valid and also mostly uncontroversial: there is no *a priori best* design paradigm, and their tradeoffs must be evaluated case by case, based on the application and functionality at hand. Moreover, our characterization of mobile code paradigms still proves useful to interpret the landscape of mobile code applications and technologies, even though both have undergone considerable changes throughout the years.

References

[1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In Vitek and Tschudin [20], pages 111–130.

[2] M. Baldi and G. P. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proc. of the 20th Int. Conf. on Software Engineering*, pages 146–155, Apr. 1998.

[3] A. Carzaniga, G. P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proc. of the 19th Int. Conf. on Software Engineering*, pages 22–32, Apr. 1997.

[4] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In Vitek and Tschudin [20], pages 93–111.

[5] M. Fitzgerald. Recasting the word processor for a connected world. *The New York Times*, Feb. 11 2007.

[6] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[7] C. Ghezzi and G. Vigna. Mobile Code Paradigms and Technologies: A Case Study. In *Proc. of the 1st Int. Workshop on Mobile Agents*, LNCS 1219, pages 39–49. Springer, 1997.

[8] J. Gosling and H. McGilton. The Java Language Environment. A White Paper, May 1996.

[9] R. Gray, G. Cybenko, D. Kotz, R. Peterson, and D. Rus. D'Agents: Applications and Performance of a Mobile-Agent System. *Software—Practice and Experience*, 32(6):543–573, May 2002.

[10] C. Harrison, D. Chess, and A. Kershenbaum. Mobile Agents: Are they a good idea? In Vitek and Tschudin [20], pages 25–47.

[11] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[12] G. P. Picco. *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility*. PhD thesis, Politecnico di Torino, Italy, Feb. 1998.

[13] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in Java. In *Proc. of 2nd Int. Symp. on Agents Systems and Applications and 4th Int. Symp. on Mobile Agents (ASA/MA)*, LNCS 1882. Springer, Sept. 2000.

[14] T. Sander and C. Tschudin. Protecting Mobile Agents Againts Malicious Hosts. In Vigna [18].

[15] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public Internet measurement facility. In *USITS '03: 4th USENIX Symposium on Internet Technologies and Systems*, pages 225–238, Mar. 2003.

[16] J. Stamos and D. Gifford. Remote Evaluation. *ACM Trans. on Programming Languages and Systems*, 12(4):537–565, Oct. 1990.

[17] D. Tennenhouse and D. Wetherall. Towards and Active Network Architecture. *Computer Communication Review*, 26(2), Apr. 1996.

[18] G. Vigna, editor. *Mobile Agents and Security*, volume 1419 of *LNCS State-of-the-Art Survey*. Springer, June 1998.

[19] G. Vigna. *Mobile Code Technologies, Paradigms, and Applications*. PhD thesis, Politecnico di Milano, Italy, Feb. 1998.

[20] J. Vitek and C. Tschudin, editors. *Mobile Object Systems: Towards the Programmable Internet*. LNCS 1222. Springer, Apr. 1997.

[21] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proc. of OPENARCH*, Apr. 1998.

[22] J. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.