Understanding Code Mobility

Alfonso Fuggetta, *Member*, *IEEE*, Gian Pietro Picco, *Member*, *IEEE*,
and Giovanni Vigna, *Member*, *IEEE*

Abstract—The technologies, architectures, and methodologies traditionally used to develop distributed applications exhibit a variety of limitations and drawbacks when applied to large scale distributed settings (e.g., the Internet). In particular, they fail in providing the desired degree of configurability, scalability, and customizability. To address these issues, researchers are investigating a variety of innovative approaches. The most promising and intriguing ones are those based on the ability of moving code across the nodes of a network, exploiting the notion of *mobile code*. As an emerging research field, code mobility is generating a growing body of scientific literature and industrial developments. Nevertheless, the field is still characterized by the lack of a sound and comprehensive body of concepts and terms. As a consequence, it is rather difficult to understand, assess, and compare the existing approaches. In turn, this limits our ability to fully exploit them in practice, and to further promote the research work on mobile code. Indeed, a significant symptom of this situation is the lack of a commonly accepted and sound definition of the term "mobile code" itself.

This paper presents a conceptual framework for understanding code mobility. The framework is centered around a classification that introduces three dimensions: technologies, design paradigms, and applications. The contribution of the paper is two-fold. First, it provides a set of terms and concepts to understand and compare the approaches based on the notion of mobile code. Second, it introduces criteria and guidelines that support the developer in the identification of the classes of applications that can leverage off of mobile code, in the design of these applications, and, finally, in the selection of the most appropriate implementation technologies. The presentation of the classification is intertwined with a review of state-of-the-art in the field. Finally, the use of the classification is exemplified in a case study.

Index Terms—Mobile code, mobile agent, distributed application, design paradigm.

1 Introduction

Computer networks are evolving at a fast pace, and this evolution proceeds along several lines. The *size* of networks is increasing rapidly, and this phenomenon is not confined just to the Internet, whose tremendous growth rate is well-known. Intra- and inter-organization networks experience an increasing diffusion and growth as well, fostered by the availability of cheap hardware and motivated by the need for uniform, open, and effective information channels inside and across the organizations. A side effect of this growth is the significant increase of the network traffic, which in turn triggers research and industrial efforts to enhance the *performance* of the communication infrastructure. Network links are constantly improved, and technological developments lead to increased computational power on both intermediate and end network nodes.

The increase in size and performance of computer networks is both the cause and the effect of an important phenomenon: networks are becoming pervasive and ubiquitous. By *pervasive*, we mean that network connectivity is no longer an expensive add-on. Rather it is a basic feature of any computing facility, and, in perspective, also of many products in the consumer electronics market (e.g., televisions). By *ubiquitous*, we refer to the ability of exploiting network connectivity independently of the physical location of the user. Developments in wireless technology free network nodes from the constraint of being placed at a fixed physical location and enable the advent of so-called *mobile computing*. In this new scenario, mobile users can move together with their hosts across different physical locations and geographical regions, still being connected to the net through wireless links.

Another important phenomenon is the increasing availability of easy-to-use technologies accessible also to naive users (e.g., the World Wide Web). These technologies have triggered the creation of new application domains and even new markets. This is changing the nature and *role* of networks, and particularly of the Internet. They cannot be considered just plain communication technologies. Nowadays, modern computer networks constitute innovative media that support new forms of cooperation and communication among users. Terms like "electronic commerce," or "Internet phone" are symptomatic of this change.

However, this evolution path is not free of obstacles and several challenging problems must be addressed. The growing size of networks raises a problem of *scalability*. Most results that are significant for small networks are often inapplicable when scaled to a world-wide network like the Internet. For instance, while it might be conceivable to apply a global snapshot algorithm to a LAN, its performance is unacceptable in an Internet setting. Wireless connectivity poses even tougher problems [1], [2]. Network nodes may move and be connected discontinuously, hence

- *A. Fuggetta and G. Vigna are with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, P.za Leonardo da Vinci, 32, I-20133 Milano, Italy. E-mail: {fuggetta, vigna}@elet.polimi.it.*
- *G.P. Picco is with the Dipartimento di Automatica e Informatica, Politecnico di Torino, C.so Duca degli Abruzzi 24, I-10129 Torino, Italy. E-mail: picco@polito.it.*

Manuscript received 16 July 1997; revised 18 Dec. 1997.
Recommended for acceptance by G.-C. Roman.

distributed systems are undermined, and we need to adapt and extend existing theoretical and technological results to this new scenario. Another relevant issue is the diffusion of network services and applications to very large segments of our society. This makes it necessary to increase the *customizability* of services, so that different classes of users are enabled to tailor the functionality and interface of a service according to their specific needs and preferences. Finally, the dynamic nature of both the underlying communication infrastructure and the market requirements demand increased *flexibility* and *extensibility*.

There have been many attempts to provide effective answers to this multifaceted problem. Most of the proposed approaches, however, try to adapt well-established models and technologies within the new setting, and usually take for granted the traditional client-server architecture. For example, CORBA [3] integrates remote procedure calls (RPCs) with the object-oriented paradigm. It attempts to combine the benefits of the latter in terms of modularity and reuse, with the well-established communication mechanism of the former. However, this approach does not ensure the degree of flexibility, customizability, and reconfigurability needed to cope with the challenging requirements discussed so far.

A different approach originates in the promising research area exploiting the notion of *mobile code*. Code mobility can be defined informally as the capability to dynamically change the bindings between code fragments and the location where they are executed [4]. The ability to relocate code is a powerful concept that originated a very interesting range of developments. However, despite the widespread interest in mobile code technology and applications, the field is still quite immature. A sound terminological and methodological framework is still missing, and there is not even a commonly agreed term to qualify the subject of this research.[1] In addition, the interest demonstrated by markets and media, due to the fact that mobile code research is tightly bound to the Internet, has added an extra level of noise, by introducing hypes and sometimes unjustified expectations. In the next section we present the main differences between mobile code and other related approaches, and the motivations and main contributions of this paper.

2 Motivations and Approach

Code mobility is not a new concept. In the recent past, several mechanisms and facilities have been designed and implemented to move code among the nodes of a network. Examples are remote batch job submission [5] and the use of PostScript [6] to control printers. The research work on distributed operating systems has followed a more structured approach. In that research area, the main problem is to support the migration of active processes and objects (along with their state and associated code) at the operating system level [7]. In particular, *process migration* concerns the transfer of an operating system process from the machine where it is running to a different one. Migration mechanisms handle the bindings between the process and its execution environment (e.g., open file descriptors and environment variables) to allow the process to seamlessly resume its execution in the remote environment. Process migration facilities have been introduced at the operating system level to achieve load balancing across network nodes. Therefore, most of these facilities provide transparent process migration: the programmer has neither control nor visibility of the migration process. Other systems provide some form of control over the migration process. For example, in Locus [8] process migration can be triggered either by an external signal or by the explicit invocation of the `migrate` system call. *Object migration* makes it possible to move objects among address spaces, implementing a finer grained mobility with respect to process-level migration. For example, Emerald [9] provides object migration at any level of granularity ranging from small, atomic data to complex objects. Emerald does not provide complete transparency since the programmer can determine objects locations and may request explicitly the migration of an object to a particular node. An example of system providing transparent migration is COOL [10], an object-oriented extension of the Chorus operating system [11]. COOL is able to move objects among address spaces without user intervention or knowledge.

Process and object migration address the issues that arise when code and state are moved among the hosts of a loosely coupled, small scale distributed system. However, they are insufficient when applied in larger scale settings. Nevertheless, the migration techniques discussed so far have been taken as a starting point for the development of a new breed of systems providing enhanced forms of code mobility. These systems, often referred to as *Mobile Code Systems* (MCSs), exhibit several innovations with respect to existing approaches:

Code mobility is exploited on an Internet-scale. Distributed systems providing process or object migration have been designed having in mind small-scale computer networks, thus assuming high bandwidth, small predictable latency, trust, and, often, homogeneity. Conversely, MCSs are conceived to operate in large scale settings where networks are composed of heterogeneous hosts, managed by different authorities with different levels of trust, and connected by links with different bandwidths (e.g., wireless slow connections and fast optical links).

Programming is location aware. Location is a pervasive abstraction that has a strong influence on both the design and the implementation of distributed applications. Mobile code systems do not paper over the location of application components, rather, applications are location-aware and may take actions based on such knowledge.

Mobility is under programmer's control. The programmer is provided with mechanisms and abstractions that enable the shipping and fetching of code fragments (or even entire components) to/from remote nodes. The underlying runtime support provides basic functionalities (e.g., data mar-

Mobility is not performed just for load balancing. Process and object migration aim at supporting load balancing and performance optimization. Mobile code systems address a much wider range of needs and requirements, such as service customization, dynamic extension of application functionality, autonomy, fault tolerance, and support for disconnected operations.

To cope with this variety of requirements and needs, industrial and academic researchers have proposed a number of MCSs. This lively and sometimes chaotic research activity has generated some confusion about the semantics of mobile code concepts and technologies.

A first problem is the unclear distinction between implementation technologies, specific applications, and paradigms used to design these applications. In an early and yet valuable assessment of code mobility [12], the authors analyze and compare issues and concepts that belong to different abstraction levels. Similarly, in a recent work about autonomous objects [13], *mechanisms* like REV [14] and RPC [15] are compared to the Echo distributed *algorithms* [16], to *applications* like "intelligent e-mail" and Web browsers, and to *paradigms* for structuring distributed applications, like mobile agents. We argue that these different concepts and notions cannot be compared directly. It is as inappropriate and misleading as trying to compare the `emacs` editor, the `fork` Unix system call, and the client-server design paradigm.

There is also confusion about terminology. For instance, several systems [17], [18] claim to be able to move the *state* of a component along with its code. This assertion is justified by the availability of mechanisms that allow the programmer to pack some portion of the data space of an executing component before the component's code is sent to a remote destination. Indeed, this is quite different from the situation where the run-time image of the component is transferred as a whole, including its execution state (i.e., program counter, call stack, etc.). In the former case, it is the programmer's task to rebuild the execution state of a component after its migration, using the data transferred with the code. Conversely, in the latter case this task is carried out by the run-time support of the MCS. Another terminological confusion stems from the excessive overload of the term "mobile agent." This term is used with different and somewhat overlapping semantics in both the distributed systems and artificial intelligence research communities. In the distributed system community the term "mobile agent" is used to denote a software component that is able to move between different execution environments. This definition has actually different interpretations. For example, while in Telescript [19] an agent is represented by a thread that can migrate among different nodes carrying its execution state, in TACOMA [17] agents are just code fragments associated with initialization data that can be shipped to a remote host. They do not have the ability to migrate once they have started their execution. On the other hand, in the artificial intelligence community the term "agent" denotes a software[2] component that is able to achieve a goal by performing actions and reacting to events in a dynamic environment [20]. The behavior of this component is determined by the knowledge of the relationships among events, actions, and goals. Moreover, knowledge can be exchanged with other agents, or increased by some inferential activity [21]. Although mobility is not the most characterizing aspect of these entities [22], there is a tendency to blend this notion of intelligent agent with the one originating from distributed systems and thus assume implicitly that a mobile agent is also intelligent (and vice versa). This is actually generating confusion since there is a mix of concepts and notions that belong to two different layers, i.e., the layer providing code mobility and the one exploiting it. Finally, there is no definition or agreement about the distinguishing characteristics of languages supporting code mobility. In [23], Knabe lists the essential characteristics of a mobile code language. They include support for manipulating, transmitting, receiving, and executing "code-containing objects." However, there is no discussion about how to manage the state of mobile components. Other contributions [24], [12] consider only the support for mobility of both code and state, without mentioning weaker forms of code mobility involving code migration alone—as we discuss later on in the paper.

Certainly, confusion and disagreement are typical of a new and still immature research field. Nevertheless, research developments are fostered not only by novel ideas, mechanisms, and systems, but also by a rationalization and conceptualization effort that re-elaborates on the raw ideas, seeking for a common and stable ground on which to base further endeavors. Research on code mobility is not an exception. The technical concerns raised by performance and security of MCSs are not the only factors hampering full acceptance and exploitation of mobile code. A conceptual framework is needed to foster understanding of the multifaceted mobile code scenario. It will enable researchers and practitioners to assess and compare different solutions with respect to a common set of reference concepts and abstractions—and go beyond it. To be effective, this conceptual framework should also provide valuable information to application developers, actually guiding the evaluation of opportunities for exploitation of code mobility during the different phases of application development.

These considerations provide the rationale for the classification presented in this paper. The classification introduces abstractions, models, and terms to characterize the different approaches to code mobility proposed so far, highlighting commonalities, differences, and applicability. The classification is organized along three dimensions that are of paramount importance during the actual development process: *technologies*, *design paradigms*, and *application domains*. Mobile code technologies are the languages and systems that provide *mechanisms* enabling and supporting code mobility. Some of these technologies have been already mentioned and are discussed in greater detail in the next section. Mobile code technologies are used by the application developer in the implementation stage. Design paradigms are the *architectural styles* that the application designer uses in defining the application architecture. An architectural style identifies a specific configuration for the

design paradigms. Application domains are *classes of applications* that share the same general goal, e.g., distributed information retrieval or electronic commerce. They play a role in defining the application requirements. The expected benefits of code mobility in a number of application domains is the motivating force behind this research field.

Our classification will break down in a vertical distinction among these three layers, as well as in an horizontal distinction among the peculiarities of the various approaches found in literature. Section 3 presents a general model and a classification of the mechanisms provided by mobile code technologies. The classification is then used to survey and characterize several MCSs. Section 4 presents mobile code design paradigms and discusses their relationships with mobile code technologies. Section 5 discusses the advantages of the mobile code approach and presents some application domains that are supposed to benefit from the use of some form of code mobility. Finally, in Section 6 we exemplify the use of the classification by applying it to a case study in the network management application domain.

3 Mobile Code Technologies

Mobile code technologies include programming languages and their corresponding run-time supports. At a first glance, these technologies provide quite different concepts and primitives. For this reason, the first part of this section introduces some reference abstractions, and then seeks out and classifies the different mechanisms that allow an application to move code and state across the nodes of a network. We are concerned here only with the issues strictly related to mobility. Other aspects of mobile code technology are indeed relevant, such as security or strategies for translation and execution. On-going work is defining a similar framework for these aspects as well. In the second part of the section (Section 3.3), the classification of mobility mechanisms is used to characterize the features provided by several existing MCSs. The classification accommodates several technologies found in literature. The set of technologies considered is not exhaustive, and is constrained by space and by the focus of the paper. However, the reader may actually verify the soundness of the classification by applying it to other MCSs not considered here, like the ones described in [25], [26], [27]. Also, the reader interested in a more detailed analysis of the linguistic problems posed by the introduction of mobility in programming languages can refer to [28], [29].

3.1 A Virtual Machine for Code Mobility

Traditional distributed systems can be accommodated in the virtual machine shown on the left-hand side of Fig. 1. The lowest layer, just upon the hardware, is constituted by the *Core Operating System* (COS). The COS can be regarded as the layer providing the basic operating system functionalities, such as file system, memory management, and process support. No support for communication or distribution is provided by this layer. Nontransparent communication services are provided by the *Network Operating System* socket services can be regarded as belonging to the NOS layer, since a socket must be opened by specifying explicitly a destination network node. The NOS, at least conceptually, uses the services provided by the COS, e.g., memory management. Network transparency is provided by the *True Distributed System* (TDS) layer. A TDS implements a platform where components, located at different sites of a network, are perceived as local. Users of TDS services do not need to be aware of the underlying structure of the network. When a service is invoked, there is no clue about the node of the network that will actually provide the service, and even about the presence of a network at all. As an example, CORBA [3] services can be regarded as TDS services since a CORBA programmer is usually unaware of the network topology and always interacts with a single well-known object broker. At least in principle, the TDS is built upon the services provided by the underlying NOS.

Technologies supporting code mobility take a different perspective. The structure of the underlying computer network is not hidden from the programmer, rather it is made manifest. In the right-hand side of Fig. 1 the TDS is replaced by *Computational Environments* (CEs) layered upon the NOS of each network host. In contrast with the TDS, the CE retains the "identity" of the host where it is located. The purpose of the CE is to provide applications with the capability to dynamically relocate their components on different hosts. Hence, it leverages off of the communication channels managed by the NOS and of the low-level resource access provided by the COS to handle the relocation of code, and possibly of state, of the hosted software components.

We distinguish the components hosted by the CE in *executing units* (EUs) and *resources*. Executing units represent sequential flows of computation. Typical examples of EUs are single-threaded processes or individual threads of a multi-threaded process. Resources represent entities that can be shared among multiple EUs, such as a file in a file system, an object shared by threads in a multi-threaded object-oriented language, or an operating system variable. Fig. 2 illustrates our modeling of EUs as the composition of a *code segment*, which provides the static description for the behavior of a computation, and a *state* composed of a *data space* and an *execution state*. The data space is the set of references to resources that can be accessed by the EU. As explained later on, these resources are not necessarily colocated with the EU on the same CE. The execution state contains private data that cannot be shared, as well as control information related to the EU state, such as the call stack and the instruction pointer. For example, a Tcl interpreter P_X executing a Tcl script X can be regarded as an EU where the code segment is X; the data space is composed of variables containing the handles for files and references to system environment variables used by P_X; the execution state is composed of the program counter and the call stack maintained by the interpreter, along with the other variables of X.

3.2 Mobility Mechanisms

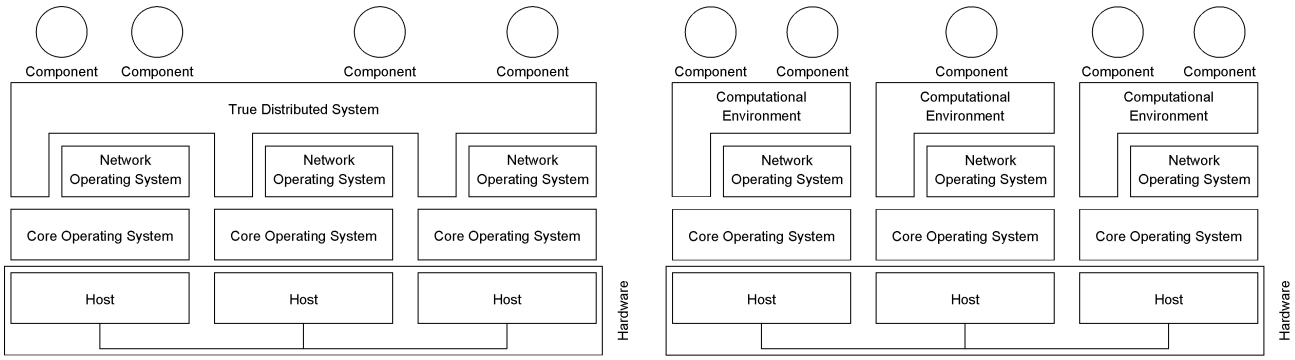

Fig. 1. Traditional systems vs. MCSs. Traditional systems, on the left-hand side, may provide a TDS layer that hides the distribution from the programmer. Technologies supporting code mobility, on the right hand side, explicitly represent the location concept, thus the programmer needs to specify *where*—i.e., in which CE—a computation has to take place.

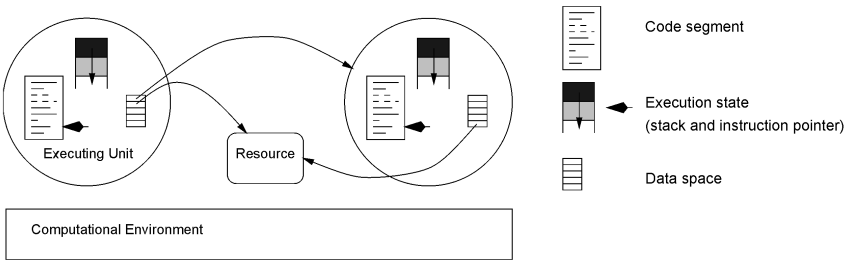

Fig. 2. The internal structure of an executing unit.

EU and its code segment is generally static. Even in environments that support dynamic linking, the code linked belongs to the local CE. This is not true for MCSs. In MCSs, the code segment, the execution state, and the data space of an EU can be relocated to a different CE. In principle, each of these EU constituents might move independently. However, we will limit our discussion to the alternatives adopted by existing systems.

The portion of an EU that needs to be moved is determined by composing orthogonal mechanisms supporting mobility of code and execution state with mechanisms for data space management. For this reason, we will analyze them separately. Fig. 3 presents a classification of mobility mechanisms.

3.2.1 Code and Execution State Mobility

Existing MCSs offer two forms of mobility, characterized by the EU constituents that can be migrated. *Strong mobility* is the ability of an MCS (called *strong MCS*) to allow migration of both the code and the execution state of an EU to a different CE. *Weak mobility* is the ability of an MCS (called *weak MCS*) to allow code transfer across different CEs; code may be accompanied by some initialization data, but no migration of execution state is involved.

Strong mobility is supported by two mechanisms: *migration* and *remote cloning*. The migration mechanism suspends an EU, transmits it to the destination CE, and then resumes it. Migration can be either proactive or reactive. In *proactive* migration, the time and destination for migration are determined autonomously by the migrating EU. In *reactive* migration, movement is triggered by a different EU that has some mechanism creates a copy of an EU at a remote CE. Remote cloning differs from the migration mechanism because the original EU is not detached from its current CE. As in migration, remote cloning can be either proactive or reactive.

Mechanisms supporting weak mobility provide the capability to transfer code across CEs and either link it dynamically to a running EU or use it as the code segment for a new EU. Such mechanisms can be classified according to the direction of code transfer, the nature of the code being moved, the synchronization involved, and the time when code is actually executed at the destination site. As for direction of code transfer, an EU can either *fetch* the code to be dynamically linked and/or executed, or *ship* such code to another CE. The code can be migrated either as *stand-alone code* or as a *code fragment*. Standalone code is self-contained and will be used to instantiate a new EU on the destination site. Conversely, a code fragment must be linked in the context of already running code and eventually executed. Mechanisms supporting weak mobility can be either *synchronous* or *asynchronous*, depending on whether the EU requesting the transfer suspends or not until the code is executed. In asynchronous mechanisms, the actual execution of the code transferred may take place either in an *immediate* or *deferred* fashion. In the first case, the code is executed as soon as it is received, while in a deferred scheme execution is performed only when a given condition is satisfied—e.g., upon first invocation of a portion of the code fragment or as a consequence of an application event.

3.2.2 Data Space Management

Upon migration of an EU to a new CE, its data space, i.e.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.