Understanding Code Mobility

Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna

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 twofold. 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 the state of the art in the field. Finally, the use of the classification is exemplified in a case study.

Keywords— **Mobile code, mobile agent, distributed application, design paradigm.**

I. 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.

A. Fuggetta and G. Vigna are with Dipartimento di Elettronica e Informazione, Politecnico di Milano, P.za Leonardo da Vinci, 32, I-20133, Italy. E-mail: {fuggetta,vigna}@elet.polimi.it.

G.P. Picco is with Dipartimento di Automatica e Informatica, Politecnico di Torino, C.so Duca degli Abruzzi 24, I-10129, Italy. E-mail: picco@polito.it.

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 the topology of the network is no longer defined statically. As a consequence, some of the basic tenets of research on 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.

II. 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 run-time support provides basic functionalities (e.g., data marshaling, code check-in, and security), but does not have any control over migration policies.

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

[1] Hereafter, we use interchangeably the terms *code mobility* and *mobile code*, although other authors prefer different terms such as *mobile computations*, *mobile object systems*, or *program mobility*.

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, and so on). 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 multi-faceted 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 components of the system and their mutual interactions. Client-server and peer-to-peer are well-known examples of 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.

[2]In this paper we ignore the implications of broader notions of agent which are not restricted to the software domain.

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 III 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 IV presents mobile code design paradigms and discusses their relationships with mobile code technologies. Section V 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 VI we exemplify the use of the classification by applying it to a case study in the network management application domain.

III. 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 III-C), the classification of mobility mechanisms is used to characterize the features provided by several existing MCSs. The classification accomodates 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].

A. A Virtual Machine for Code Mobility

Traditional distributed systems can be accommodated in the virtual machine shown on the left-hand side of Figure 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. Non-transparent communication services are provided by the *Network Operating System* (NOS) layer. Applications using NOS services address explicitly the host targeted by communication. For instance, 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 Figure 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. Figure 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 co-located 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.

B. Mobility Mechanisms

In conventional systems, each EU is bound to a single CE for its entire lifetime. Moreover, the binding between the EU and its code segment is generally static. Even in environments that support dynamic linking, the code linked be-

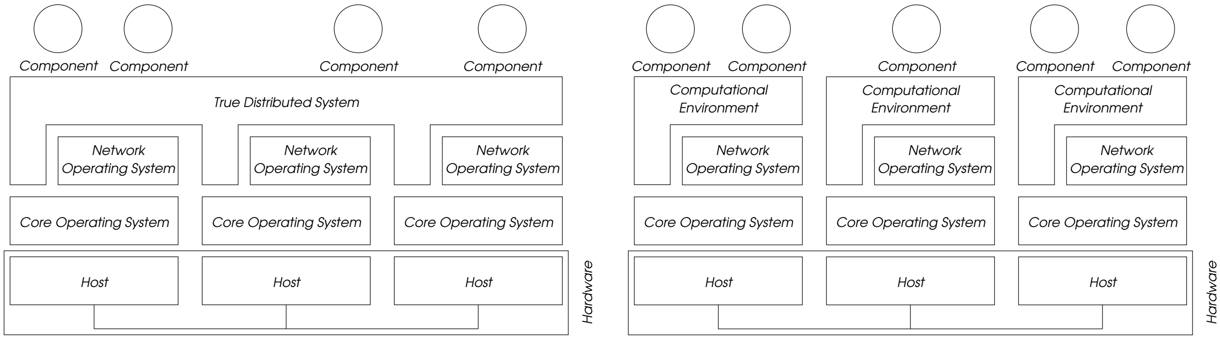

Fig. 1. Traditional systems vs. MCSs. Traditional systems, on the left hand side, may provide a TDS layer that hides the distribution from the programmer. Technologies supporting code mobility, on the right hand side, explicitly represent the location concept, thus the programmer needs to specify *where*—i.e.,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

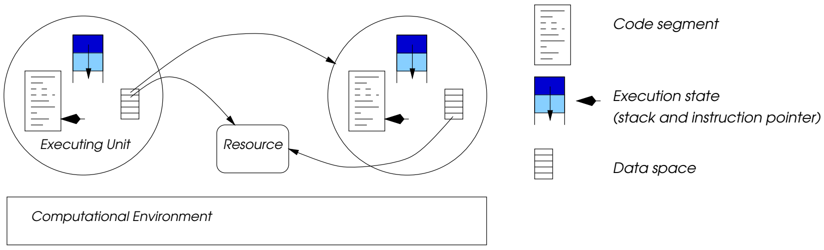

Fig. 2. The internal structure of an executing unit.

longs 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. Figure 3 presents a classification of mobility mechanisms.

B.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 kind of relationship with the EU to be migrated, e.g., an EU acting as a manager of roaming EUs. The remote cloning 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*. Stand-alone 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.

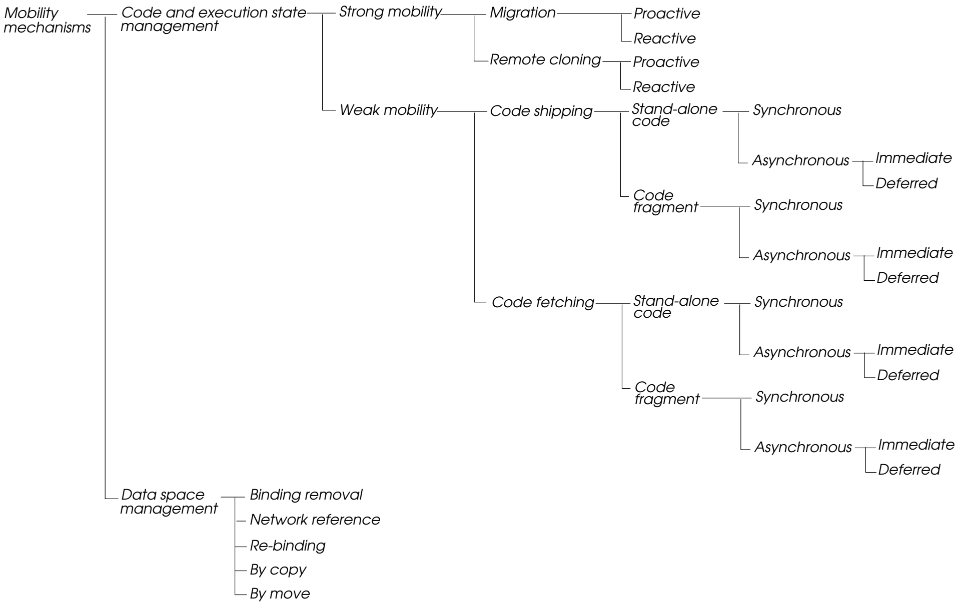

Fig. 3. A classification of mobility mechanisms.

B.2 Data Space Management

Upon migration of an EU to a new CE, its data space, i.e. the set of bindings to resources accessible by the EU, must be rearranged. This may involve voiding bindings to resources, re-establishing new bindings, or even migrating some resources to the destination CE along with the EU. The choice depends on the nature of the resources involved, the type of binding to such resources, as well as on the requirements posed by the application.

We model resources as a triple $Resource = \langle I, V, T \rangle$, where I is a unique identifier, V is the value of the resource, and T is its type, which determines the structure of the information contained in the resource as well as its interface. The type of the resource determines also whether the resource is *transferrable* or *not transferrable*, i.e. whether, in principle, it can be migrated over the network or not. For example, a resource of type "stock data" is likely to be transferrable, while a resource of type "printer" probably is not. Transferrable resource instances can be marked as *free* or *fixed*. The former can be migrated to another CE, while the latter are associated permanently with a CE. This characteristic is determined according to application requirements. For instance, even if it might be conceivable to transfer a huge file or an entire database over the network, this might be undesirable for performance reasons. Similarly, it might be desirable to prevent transfer of classified resources, even independently of performance considerations.

Resources can be bound to an EU through three forms of binding, which constrain the data space management mechanisms that can be exploited upon migration. The strongest form of binding is *by identifier*. In this case, the EU requires that, at any moment, it must be bound to a given uniquely identified resource. Binding by identifier is exploited when an EU requires to be bound to a resource that cannot be substituted by some other equivalent resource.

A binding established *by value* declares that, at any moment, the resource must be compliant with a given type and its value cannot change as a consequence of migration. This kind of binding is usually exploited when an EU is interested in the *contents* of a resource and wants to be able to access them locally. In this case, the identity of the resource is not relevant, rather the migrated resource must have the same type and value of the one present on the source CE.

The weakest form of binding is *by type*. In this case, the EU requires that, at any moment, the bound resource is compliant with a given type, no matter what its actual value or identity are. This kind of binding is exploited typically to bind resources that are available on every CE, like system variables, libraries, or network devices. For example, if a roaming EU needs to access the local display of a machine to interact with the user through a graphical interface, it may exploit a binding with a resource of type "display". The actual value and identifier of the resource are not relevant, and the resource actually bound is determined by the current CE. Note that it is possible to have different types of binding to the *same* resource. In the example above, suppose that the roaming EU, in addition to interact with the local user through the display, needs to report progress back to the user that "owns" the EU. This is accomplished by creating, at startup, a binding by identifier to the display of the owner and a binding by type to the same resource. As we will explain shortly, after the first migration the bindings will be reconfigured so that the binding by identifier will retain its association with the owner's display, while the binding by type will be associated with the display on the destination CE.

The above discussion highlights two classes of problems that must be addressed by data space management mechanisms upon migration of an EU: resource relocation and binding reconfiguration. The way existing mechanisms tackle these problems is constrained both by the nature of the resources involved and the forms of binding to such resources. These relationships are analyzed hereafter and summarized in Table I.

Let us consider a migrating executing unit U whose data space contains a binding B to a resource R. A first general mechanism, which is independent of the type of binding or resource is *binding removal*. In this case, when U migrates, B is simply discarded. If access to bound resources must be preserved, different mechanisms must be exploited.

If U is bound to R *by identifier*, two data space management mechanisms are suitable to preserve resource identity. The first is relocation *by move*. In this case, R is transferred, along with U to the destination CE and the binding is not modified (Figure 4a). Clearly, this mechanisms can be exploited only if R is a free transferrable resource. Otherwise, a *network reference* mechanism must be used. In this case, R is not transferred and once U has reached its target CE, B is modified to reference R in the source CE. Every subsequent attempt of U to access R through B will involve some communication with the source CE (Figure 4b). The creation of inter-CE bindings is often not desirable because it exposes U to network problems—e.g., partitioning, or delays—and makes it difficult to manage state consistency since the data space is actually *distributed* over the network. On the other hand, moving away a resource from its CE may cause problems to other EUs that own bindings to the moved resource. This latter situation may be managed in different ways. A first approach is to apply binding removal, i.e., to void bindings to the resource moved (see top of Figure 4a). Subsequent attempts to access the resource through such bindings will rise an exception. A second approach is to retain the bindings to the resource at its new location by means of network references (see bottom of Figure 4a).

If B is *by value* and R is transferrable, the most convenient mechanism is data space management *by copy* because the identity of the resource is not relevant. In this case, a copy R' of R is created, the binding to R is modified to refer to R', and then R' is transferred to the destination CE along with U (see Figure 4c). Management *by move* satisfies the requirements posed by bindings by value but, in some cases, may be less convenient. In fact, in this case R would be removed from the source CE and other EUs owning bindings to R would have to cope with this event. If R cannot be transferred, the use of the network reference mechanisms is the only viable solution, with the drawbacks described previously.

If U is bound to R *by type*, the most convenient mechanism is *re-binding*. In this case B is voided and re-established after migration of U to another resource R' on the target CE having the same type of R (Figure 4d). Re-binding exploits the fact that the only requirement posed by the binding is the type of the resource, and avoids resource transfers or the creation of inter-CE bindings. Clearly, this mechanism requires that, at the destination site, a resource of the same type of R exists. Otherwise, the other mechanisms can be used depending on the type and characteristics of the resource involved.

The existing MCSs exploit different strategies as far as data space management is concerned. The nature of the resource and the type of binding is often determined by the language definition or implementation, rather than by the application programmer, thus constraining the mechanisms exploited. For instance, files are usually considered a fixed unique resource, and migration is usually managed by voiding the corresponding bindings, although files in principle could be migrated along with an EU. Replicated resources are often provided as built-in to provide access to system features in a uniform way across all CEs. The next section will provide more insights about mobility mechanisms in existing MCSs.

C. A Survey of Mobile Code Technologies

Currently available technologies differ in the mechanisms they provide to support mobility. In this section we apply the classification of mobility mechanisms presented so far to a number of existing MCSs.

C.1 Agent Tcl

Developed at the University of Darthmouth, Agent Tcl [30] provides a Tcl interpreter extended with support for strong mobility. In Agent Tcl, an EU (called *agent*) is implemented by a Unix process running the language interpreter. Since EUs run in separate address spaces, they can share only resources provided by the underlying operating system, like files. Such resources are considered as not transferrable. The CE abstraction is implemented by the operating system and the language run-time support. In Agent Tcl, EUs can `jump` to another CE, `fork` a new EU at a remote CE, or `submit` some code to a remote CE. In the first case, a proactive migration mechanism enables movement of a whole Tcl interpreter along with its code and execution state. In the second case, a proactive remote cloning mechanism is implemented. In both cases, bindings in the data space of a migrating EU are removed. In the third case, a code shipping mechanism for stand-alone code is exploited to perform remote execution of a Tcl script in a newly created EU at the destination CE. This mechanism is asynchronous and immediate. A copy of the variables belonging to the execution state of the EU invoking the `submit` may be passed as parameters of this operation in order to migrate these variables together with the Tcl script.

C.2 Ara

Developed at University of Kaiserslautern, Ara [24] is a multi-language MCS that supports strong mobility. Ara EUs, called *agents*, are managed by a language-independent system core plus interpreters for the languages supported—at the time of writing C, C++, and Tcl. The core and the interpreters constitute the CE, whose services are made accessible to agents through the *place* abstraction. Mobility is supported through proactive migration,

	Free Transferrable	*Fixed Transferrable*	*Fixed Not Transferrable*
By Identifier	By move (Network reference)	Network reference	Network reference
By Value	By copy (By move, Network reference)	By copy (Network reference)	(Network reference)
By Type	Re-binding (Network reference, By copy, By move)	Re-binding (Network reference, By copy)	Re-binding (Network reference)

TABLE I

Bindings, resources and data space management mechanisms.

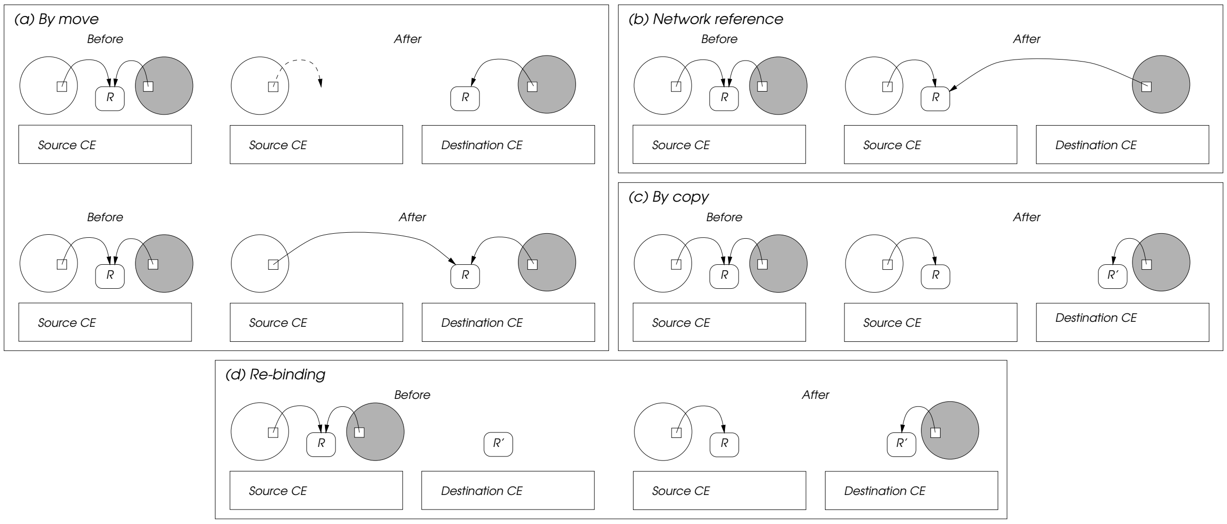

Fig. 4. Data space management mechanisms. For each mechanism, the configuration of bindings before and after migration of the grayed EU is shown.

and data space management is simplified by the fact that agents cannot share anything but system resources—whose bindings are always removed upon migration.

C.3 Facile

Developed at ECRC in Münich, Facile [31] is a functional language that extends the Standard ML language with primitives for distribution, concurrency, and communication. The language has been extended further in [23] to support weak mobility. Executing units are implemented as threads that run in Facile CEs, called *nodes*. The *channel* abstraction is used for communication between threads. Channels can be used to communicate any legal value of the Facile language. In particular, functions may be transmitted through channels since they are first-class language elements. Communication follows the rendez-vous model: both the sender and the receiver are blocked until communication takes place. For this reason, mobility mechanisms can be regarded as supporting both code shipping and code fetching—depending on whether an EU is a sender or a receiver. In addition, the programmer can specify whether the function transmitted has to be considered as stand-alone code or as a code fragment. When the function has been transferred, the communication channel is closed, and the receiver EU is free to evaluate the function received or defer its evaluation. Therefore, the mechanism is asynchronous and supports both immediate and deferred execution. As for data space management, this takes place always by copy, except for special variables called *ubiquitous values*. They represent resources replicated in each Facile node and are always accessed with bindings by type, exploiting a re-binding mechanism.

C.4 Java

Developed by Sun Microsystems, Java [32] has triggered most of the attention and expectations on code mobility. The original goal of the language designers was to provide a portable, clean, easy-to-learn, and general-purpose object-oriented language, which has been subsequently re-targeted by the growth of Internet. The Java compiler translates Java source programs into an intermediate, platform-independent language called *Java Byte Code*. The byte code is interpreted by the *Java Virtual Machine* (JVM)—the CE implementation. Java provides a programmable mechanism, the *class loader*, to retrieve and link dynamically classes in a running JVM. The class loader is invoked by the JVM run-time when the code currently in execution contains an unresolved class name. The class loader actually retrieves the corresponding class, possibly from a remote host, and then loads the class in the JVM. At this point, the corresponding code is executed. In addition, class downloading and linking may be triggered explicitly by the application, independent of the need to execute the class code. Therefore Java supports weak mobility using mechanisms for fetching code fragments. Such mechanisms are asynchronous and support both immediate and deferred execution. In both cases, the code loaded is always executed from scratch and has neither execution state nor bindings to resources at the remote host—no data space management is needed.

One of the key success factors of Java is its integration with World Wide Web technology. Web browsers have been extended to include a JVM. Java classes called *applets* can be downloaded along with HTML pages to allow for active presentation of information and interactive access to a server. From the viewpoint we took in our classification, we regard this as a particular *application* of mobile code technology. However, it can be argued also that the combination of a Web browser and a JVM is so frequent that it can be regarded as a technology per se, conceived explicitly for the development of Web applications. From this perspective, the presence of a JVM is hidden and its mechanisms are used to provide a higher-level layer where browsers constitute the CEs and applets are EUs executing concurrently within them. In this context, the downloading of applets can be regarded as a mechanism provided by the browser to support fetching of stand-alone code.

C.5 Java Aglets

The Java Aglets API (J-AAPI) [33], developed by IBM Tokyo Research Laboratory in Japan, extends Java with support for weak mobility. Aglets [34], the EUs, are threads in a Java interpreter which constitutes the CE. The API provides the notion of *context* as an abstraction of the CE. The context of an aglet provides a set of basic services, e.g., retrieval of the list of aglets currently contained in that context or creation of new aglets within the context. Java Aglets provides two migration primitives: `dispatch` is the primitive that performs code shipping of stand-alone code (the code segment of the aglet) to the context specified as parameter. The mechanism is asynchronous and immediate. The symmetrical primitive `retract` performs code fetching of stand-alone code, and is used to force an aglet to come back to the context where `retract` is executed, with a synchronous, immediate mechanism. In both cases, the aglet is re-executed from scratch after migration, although it retains the value of its object attributes which are used to provide an initial state for its computation. The attribute values may contain references to resources, which are always managed by copy. Finally, being based on Java, the Aglets API supports Java mechanisms as well.

C.6 M0

Implemented at the University of Geneva, M0 [35] is a stack-based interpreted language that implements the concept of *messengers*. Messengers—representing EUs—are sequences of instructions that are transmitted among *platforms*—representing CEs—and executed unconditionally upon receipt. Messengers [36], in turn, can *submit* the code of other messengers to remote platforms. Resources are always considered transferrable and fixed, and the submitting messenger may copy them in the message containing the submitted code to make them available at the destination CE. Therefore, M0 is a weak MCS providing shipping of stand-alone code (whose execution is asynchronous and immediate), and data space management is by copy.

C.7 Mole

Developed at University of Stuttgart, Mole [37], [38] is a Java API that supports weak mobility. Mole agents are Java objects which run as threads of the JVM, which is abstracted into a *place*, the Mole CE. A place provides access to the underlying operating system through *service agents* which, differently from *user agents*, are always stationary. Shipping of stand-alone code is provided with an asynchronous, immediate mechanism. The code and data to be sent are determined automatically upon migration using the notion of island [39]. An island is the transitive closure over all the objects referenced by the main agent object. Islands, which are generated automatically starting from the main agent object, cannot have object references to the outside; inter-agent references are symbolic and become void upon migration. Hence, data space management by move is exploited.

C.8 Obliq

Developed at DEC, Obliq [40] is an untyped, object-based, lexically scoped, interpreted language. Obliq allows for remote execution of procedures by means of *execution engines* which implement the CE concept. A thread, the Obliq EU, can request the execution of a procedure on a remote execution engine. The code for such procedure is sent to the destination engine and executed there by a newly created EU. The sending EU suspends until the execution of the procedure terminates. Thus, Obliq supports weak mobility using a mechanism for synchronous shipping of stand-alone code. Obliq objects are transferrable fixed resources, i.e., they are bound for their whole lifetime to the CE where they are created even if in principle they could be moved across CEs. When an EU requests the execution of a procedure on a remote CE, the references to the local objects used by the procedure are automatically translated into network references.

C.9 Safe-Tcl

Initially developed by the authors of the Internet MIME standard, Safe-Tcl [41] is an extension of Tcl [42] conceived to support active e-mail. In active e-mail, messages may include code to be executed when the recipient receives or reads the message. Hence, in Safe-Tcl there are no mobility or communication mechanisms at the language level—they must be achieved using some external support, like e-mail. Rather, mechanisms are provided to protect the recipient's CE, which is realized following a *twin interpreter* scheme. The twin interpreter consists of a *trusted* interpreter, which is a full-fledged Tcl interpreter, and an *untrusted* interpreter, whose capabilities have been restricted severely, so that one can execute code of uncertain origin without being damaged. The owner of the interpreter may decide to export procedures which are guaranteed to be safe from the trusted interpreter to the untrusted one. Presently, most of the fundamental features of Safe-Tcl have been included in the latest release of Tcl/Tk, and a plug-in for the Netscape browser has been developed, allowing Safe-Tcl scripts to be included in HTML pages [43], much like Java applets.

C.10 Sumatra

Sumatra [44], developed at University of Maryland, is a Java extension designed expressly to support the implementation of *resource-aware* mobile programs, i.e. programs which are able to adapt to resource changes by exploiting mobility. Sumatra provides support for strong mobility of Java threads, which are Sumatra EUs. Threads are executed within *execution engines*, i.e. dynamically created interpreters which extend the abstract machine provided by the JVM with methods that embody proactive migration mechanisms, proactive remote cloning, and shipping of stand-alone code with synchronous, immediate execution. Threads or stand-alone code can be migrated separately from the objects they need. The *object-group* abstraction is provided to represent dynamically created object aggregates that determine the unit of mobility as well as the unit of persistency. Objects belonging to a group must be explicitly checked in and out, and thread objects cannot be checked in an object-group. The rationale for the absence of an automatic mechanism is to give the programmer the ability to modify dynamically the granularity of the unit of mobility. Data space management in an object-group is always by move; bindings to migrated objects owned by EUs in the source CE are transformed into network references.

C.11 TACOMA

In TACOMA [17] (Tromsø And COrnell Mobile Agents), the Tcl language is extended to include primitives that support weak mobility. Executing units, called *agents*, are implemented as Unix processes running the Tcl interpreter. The functionality of the CE is implemented by the Unix operating system plus a dedicated run-time supporting agent check-in and check-out. Code shipping of stand-alone code is supported by mechanisms providing both synchronous and asynchronous immediate execution. Initialization data for the new EU are encapsulated in a data structure called *briefcase*, while resources in the CE are contained in stationary data structures called *cabinets*. Upon migration, data space management by copy can be exploited to provide the new EU with a resource present within the source CE cabinet. In version 1.2, the system has been extended to support a number of interpreted languages, namely Python, Scheme, Perl, and C.

C.12 Telescript

Developed by General Magic, Telescript [19] is an object-oriented language conceived for the development of large distributed applications. Security has been one of the driving factors in the language design, together with a focus on strong mobility. Telescript employs an intermediate, portable language called *Low Telescript*, which is the representation actually transmitted among *engines*, the Telescript CEs. Engines are in charge of executing *agents* and *places*, that are the Telescript EUs. Agents can move by using the `go` operation, which implements a proactive migration mechanism. A `send` operation is also available which implements proactive remote cloning. Places are stationary EUs that can contain other EUs. Data space management is ruled by the *ownership* concept which associates each resource with an owner EU. Upon migration, this information is used to determine automatically the set of objects that must be carried along with the EU. Data space management always exploits management by move for the migrating EU. Bindings to migrated resources owned by other EUs in the source site are always removed.

IV. Design Paradigms

Mobile code technologies are only one of the ingredients needed to build a software system. Software development is a complex process where a variety of factors must be taken into account: technology, organization, and methodology. In particular, a very critical issue is the relationship between technology and methodology. This relationship is often ignored or misinterpreted. Quite often, researchers and practitioners tend to believe that a technology inherently induces a methodology. Thus "it is sufficient to build good development tools and efficient languages". This is particularly evident in a critical phase of software development: software design. The goal of design is the creation of a software architecture, which can be defined as the decomposition of a software system in terms of software components and interactions among them [45]. Software architectures with similar characteristics can be represented by *architectural styles* [46] or *design paradigms*, which define architectural abstractions and reference structures that may be instantiated into actual software architectures. A design paradigm is not necessarily induced by the technology used to develop the software system—it is a conceptually separate entity. This distinction is not merely philosophical: the evolution of programming languages has clearly emphasized the issue. It is even possible for a *modular* system to be built using an assembly language, and at the same time, the adoption of sophisticated languages such as Modula-2 does not guarantee *per se* that the developed system will be really modular. Certainly, specific features of a language can be particularly well-suited to guarantee some program property, but a "good" program is not just the direct consequence of selecting a "good" language.

Traditional approaches to software design are not sufficient when designing large scale distributed applications that exploit code mobility and dynamic reconfiguration of software components. In these cases, the concepts of location, distribution of components among locations, and migration of components to different locations need to be taken explicitly into account during the design stage. As stated in [47], interaction among components residing on the same host is remarkably different from the case where components reside on different hosts of a computer network in terms of latency, access to memory, partial failure, and concurrency. Trying to paper over differences between local and remote interactions can lead to unexpected performance and reliability problems after the implementation phase.

It is therefore important to identify reasonable design paradigms for distributed systems exploiting code mobility[3], and to discuss their relationships with the technology that can be used to implement them. It is also important to notice that each of the languages mentioned in the previous section embodies mechanisms that can be used to implement one or more design paradigms. On the other hand, the paradigms themselves are independent of a particular technology, and could even be implemented without using mobile technology at all, as described in the case study presented in [49].

A. Basic Concepts

Before introducing design paradigms we present some basic concepts that are an abstraction of the entities that constitute a software system, such as files, variable values, executable code, or processes. In particular, we introduce three architectural concepts: *components*, *interactions*, and *sites*.

Components are the constituents of a software architecture. They can be further divided into *code components*, that encapsulate the know-how to perform a particular computation, *resource components*, that represent data or devices used during the computation, and *computational components*, that are active executors capable to carry out a computation, as specified by a corresponding know-how. *Interactions* are events that involve two or more components, e.g., a message exchanged among two computational components. *Sites* host components and support the execution of computational components. A site represents the intuitive notion of location. Interactions among components residing at the same site are considered less expensive than interactions taking place among components located in different sites. In addition, a computation can be actually carried out only when the know-how describing the computation, the resources used during the computation, and the computational component responsible for execution are located *at the same site*.

Design paradigms are described in terms of interaction patterns that define the relocation of and coordination among the components needed to perform a service. We will consider a scenario where a computational component A, located at site S_A needs the results of a service. We assume the existence of another site S_B, which will be involved in the accomplishment of the service.

We identify three main design paradigms exploiting code mobility: *remote evaluation*, *code on demand*, and *mobile agent*. These paradigms are characterized by the location of components before and after the execution of the service, by the computational component which is responsible for execution of code, and by the location where the computation of the service actually takes place (see Table II).

The presentation of the paradigms is based on a metaphor 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 (movable resources), 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 final effects.

A.1 Client-Server (CS)

Louise would like to have a chocolate cake, but she doesn't know the recipe, and she does not have at home either the required ingredients or an oven. Fortunately, she knows that her friend Christine knows how to make a chocolate cake, and that she has a well supplied kitchen at her place. Since Christine is usually quite happy to prepare cakes on request, Louise phones her asking: "Can you make me a chocolate cake, please?". Christine makes the chocolate cake and delivers it back to Louise.

The client-server paradigm is well-known and widely used. 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 at S_A, requests the execution of a service with an interaction with the server component B. As a response, B performs the requested service 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.

A.2 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. 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.

In the REV paradigm[4], a component A has the know-how necessary to perform the service but it lacks the resources required, 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. B, in turn,

[3]The reader interested in the original formulation of the paradigms described here is directed to [4]. A case study centered around a formalization of these paradigms using the UNITY notation is also provided in [48].

[4]Hereafter, by "remote evaluation" we will refer to the design paradigm presented in this section. Although it has been inspired by work on the REV system [14], 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 resource B	A	know-how resource **B**
Remote Evaluation	know-how A	resource B	A	*know-how* resource **B**
Code on Demand	resource A	know-how B	resource *know-how* **A**	B
Mobile Agent	know-how A	resource	—	*know-how* resource ***A***

TABLE II

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.

executes the code using the resources available there. An additional interaction delivers the results back to A.

A.3 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.

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

A.4 Mobile agent (MA)

Louise wants to prepare a chocolate cake. She has the right recipe and ingredients, but she does not have 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 batter and then goes to Christine's home, where she bakes the cake.

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. 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 since 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 is moved to a remote site, along with its state, the code it needs, and some resources required to perform the task.

B. Discussion and Comparison

The mobile code design paradigms introduced in the previous sections define a number of abstractions for representing the bindings among components, locations, and code, and their dynamic reconfiguration. Our initial experience in applying the paradigms [50], [49] suggests that these abstractions are effective in the design of distributed applications. Furthermore, they are fairly independent of the particular language or system in which they are eventually implemented.

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 types 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 an interaction involving communication through the 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. By changing their location, components may change dynamically 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. In addition, the COD paradigm enables computational components to retrieve code from other remote components, providing a flexible way to extend dynamically their behavior and the types of interaction they support.

Flexibility and dynamicity are useful, but it is not clear when these paradigms should be used, and how one can choose the right paradigm in designing a distributed application. In our opinion there is no paradigm that is the best in absolute terms. In particular the mobile code paradigms we described do not necessarily prove to be better suited for a particular application with respect to the traditional ones. The choice of the paradigms to exploit must be performed on a case-by-case basis, according to the specific type of application and to the particular functionality being designed within the application. For each case, some parameters that describe the application behavior have to be chosen, along 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, an analysis should be carried out to determine which paradigm optimizes the chosen criteria. This phase cannot take into account all the characteristics and constraints, that 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. A case study that provides guidelines on how such analysis can be carried out is given in Section VI.

Once an application has been designed, developers are faced with the choice of a suitable technology for its implementation. Even if technologies are somewhat orthogonal with respect to paradigms, some technologies are better suited to implement application designed according to particular paradigms. For example, one can implement an application designed following the REV paradigm with a technology that allows EUs to exchange just messages. In this case, the programmer has the burden to translate the code to be shipped to the remote site into the data format used in message payloads. Moreover, the receiving EU has to explicitly extract the code and invoke an intepreter in order to execute it. A mobile code technology providing mechanisms for code shipping would be more convenient and would manage marshaling, shipping, and remote interpretation tasks at the system level.

A common case is represented by the use of a weak MCS that allows for code shipping for implementing applications designed following the MA paradigm [51]. In this case, the architectural concept of a moving component must be implemented using a technology that does not preserve the execution state upon migration. Therefore the programmer has to build explicitly some appropriate data structures that allows for saving and restoring the execution state of the component in case of migration. Upon migration, the EU has to pack such data structures and send them along with the code to the remote location; then the original EU terminates. When the new EU is started on the remote CE to execute the code, it must use explicitly the encoded representation of the component's state to reconstruct, at the program level, the component's execution state. If a strongly mobile technology is used, the component can be directly mapped into a migrating EU and mobility is reduced to a single instruction. Therefore the programmer is set free from handling the management of the component's state and can concentrate on the problem to solve. A case study that analyzes these relationships in detail can be found in [49].

V. Mobile Code Applications

At the time of writing, applications exploiting code mobility can still be considered as relegated to a niche, at least if compared to traditional client-server based applications. This is a consequence of the immaturity of technology—mostly as far as performance and security [52] are concerned—and of the lack of suitable methodologies for application development. Nevertheless, the interest in mobile code is not motivated by the technology per se, rather by the benefits that it is supposed to provide by enabling new ways of building distributed applications and even of creating brand new applications. The advantages expected from the introduction of mobile code into distributed applications are particularly appealing in some specific application domains. This fact has sometimes led to identifying entire application classes with terms like "mobile agent systems" or "Internet agents" that refer more to how the applications are structured rather than to the functionality they implement. Therefore, in order to understand mobile code it is important to distinguish clearly between an application (e.g., a system to control a remote telescope) and the paradigm used to design it (e.g., the REV paradigm to identify control modules that are sent to the remote telescope) or the technology used to implement it (e.g., Java Aglets).

Hence, the purpose of this section is to provide the reader both with a grasp on the key benefits which mobile code is expected to bring, and with a non-exhaustive review of application domains which are being identified by researchers in the field as suitable for the exploitation of mobile code. This completes our conceptual framework and provides the reader with a path from the problem to the implementation, spanning application, design, and technology issues. Section VI will show an example of how our framework can be leveraged off in the network management application domain.

A. Key Benefits of Mobile Code

A major asset provided by code mobility is that it enables *service customization*. In conventional distributed systems built following a CS paradigm, servers provide 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 ability to request the remote execution of code, by converse, helps increase server flexibility without affecting permanently the size or complexity of the server. In this case, in fact, the server actually provides very simple and low-level services that seldom need to be changed. These services are then composed by the client to obtain a customized high-level functionality that meets the specific client's needs.

Mobile code is proving useful in supporting the last phases of the software development process, namely, *deployment* and *maintenance*. Software engineering addressed 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 better development tools. In a distributed setting, however, the action of installing or rebuilding the application at each site still has to be performed locally and with human intervention. Some products, notably some Web browsers, already use some limited form of program downloading to perform automatic upgrade over the Internet. Mobile code helps in providing more sophisticated automation for the installation process. For instance, a scheme could be devised where installation actions (that, by their nature, can usually be automated) are coded in a mobile program roaming across a set of hosts. There, the program could analyze the features of the local platform and operate the correct configuration and installation steps. Pushing even further these concepts, let us suppose that a new functionality is needed by an application, say, a new dialog box must be shown when a particular button is pushed on the user interface. 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 could be lenghty and, even worse, if the functionality is not fundamental for application operativity there is no guarantee that it will be actually used. In this respect, the ability to request on demand the dynamic linking of the code fragment implementing the new functionality provides several benefits. First, all changes would be centralized in the code server repository, where the last version is always present and consistent. 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 of the code to the central repository. Hence, changes could be propagated in a lazy way, concentrating the upgrade effort only where it is really needed.

Mobile code concepts and technology embody also a notion of *autonomy* of application components. Autonomy is a useful property for applications that use a heterogeneous communication infrastructure where the nodes of a network may be connected by a variety of physical links with different performances. These differences must be taken into account since the design stage. For instance, recent developments in mobile computing evidenced that low-bandwidth and low-reliable communication channels require new design methodologies for applications in a mobile setting [1], [2]. In networks where some regions are connected through wireless links while others are connected through conventional links the design becomes complex. It is important to cope with frequent disconnections and avoid the generation of traffic over the low-bandwidth links as much as possible. The CS paradigm has a unique alternative to achieve this objective: 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, which are performed locally on the server without involving communication over the physical link. Nevertheless, this solution may be impossible to achieve in certain cases given the specific application requirements. In any case, it leads to increased complexity and size, as well as reduced flexibility of the server. Code mobility overcomes these limits because it allows for specifying complex computations that are able to move across a network. This way, the services that need to be executed by a server residing in a portion of the network reachable only through an unreliable and slow link could be described in a program. This should pass once through the wireless link and be 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.

Autonomy of application components brings improved *fault tolerance* as a side-effect. In conventional client-server systems, the state of the computation is distributed between the client and the server. A client program is made of statements that are executed in the local environment, interleaved with statements that invoke remote services on the server. The server contains (copies of) data that belong to the environment of the client program, and will eventually return a result that has to be inserted into the same environment. This structure leads to well-known problems in presence of partial failures, because it is very difficult to determine where and how to intervene to reconstruct a consistent state. The action of *migrating* code, and possibly sending back the results, is not immune from this problem. In order to determine whether the code has been received and avoid duplicates or lost mobile code, an appropriate protocol must be in place. However, the action of *executing* code that embodies a set of interactions that should otherwise take place across the network is actually immune from partial failure. An autonomous component encapsulates all the state involving a distributed computation, and can be easily traced, checkpointed, and possibly recovered *locally*, without any need for knowledge of the global state.

Another advantage that comes from the introduction of code mobility in a distributed application is *data management flexibility* and *protocol encapsulation*. In conventional systems, when data are exchanged among components belonging to a distributed application, each component owns the code describing the protocol necessary to interpret the data correctly. However, it is often the case for the "know-how" related to the data to change frequently or to be determined case by case according to some external condition—thus making impractical to hard-wire the corresponding code into the application components. Code mobility enables more efficient and flexible solutions. For example, if protocols are only seldom modified and are loosely coupled with data, an application may download the code that implements a particular protocol only when the data involved in the computation need a protocol unknown to the application. Instead, if protocols are tightly coupled with the data they accompany, components could exchange messages composed by both the data and the code needed to access and manage such data.

B. Application Domains for Mobile Code

The following review of application domains for mobile code serves two purposes. First, we want to describe some of the domains which are expected to exploit in the near future the benefits described previously, in order to provide the reader with an idea of the applicability of the concepts presented so far. Second, we want to point out that some concepts which are often associated *tout court* with code mobility are not mobile code approaches per se, rather they are examples of the exploitation of mobile code in a given application domain.

B.1 Distributed Information Retrieval

Distributed information retrieval applications gather information matching some specified criteria from a set of information sources dispersed in the network. The information sources to be visited can be defined statically or determined dynamically during the retrieval process. This is a wide application domain, encompassing very diverse applications. For instance, the information to be retrieved might range from the list of all the publications of a given author to the software configuration of hosts in a network. Code mobility could improve efficiency by migrating the code that performs the search process close to the (possibly huge) information base to be analyzed [53]. This type of application has been often considered "the killer application" motivating a design based on the MA paradigm. However, analysis to determine the network traffic in some typical cases evidenced that, according to the parameters of the application, the CS paradigm sometimes can still be the best choice [4].

B.2 Active Documents

In active documents applications, traditionally passive data, like e-mail or Web pages, are enhanced with the capability of executing programs which are somewhat related with the document contents, enabling enhanced presentation and interaction. Code mobility is fundamental for these applications since it enables the embedding of code and state into documents and supports the execution of the dynamic contents during document fruition. A paradigmatic example is represented by an application that uses graphic forms to compose and submit queries to a remote database. The interaction with the user is modeled by using the COD paradigm, i.e., the user requests the active document component to the server and then performs some computation using the document as an interface. This type of application can be easily implemented by using a technology that enables fetching of remote code fragments. A typical choice is a combination of WWW technology and Java applets.

B.3 Advanced Telecommunication Services

Support, management, and accounting of advanced telecommunication services like videoconference, video on demand, or telemeeting, require a specialized "middleware" providing mechanisms for dynamic reconfiguration and user customization—benefits provided by code mobility. For example, the application components managing the setup, signalling, and presentation services for a videoconference could be dispatched to the users by a service broker. Examples of approaches exploiting code mobility can be found in [54] and [55]. A particular class of advanced telecommunications services are those supporting mobile users. In this case, as discussed earlier, autonomous components can provide support for disconnected operations, as discussed in [56].

B.4 Remote Device Control and Configuration

Remote device control applications are aimed at configuring a network of devices and monitoring their status. This domain encompasses several other application domains, e.g., industrial process control and network management. In the classical approach, monitoring is achieved by polling periodically the resource state. Configuration is performed using a predefined set of services. This approach, based on the CS paradigm, can lead to a number of problems [57]. Code mobility could be used to design and implement monitoring components that are co-located with the devices being monitored and report events that represent the evolution of the device state. In addition, the shipment of management components to remote sites could improve both performance and flexibility [50], [58]. A case study focused on the application of our taxonomy to the network management application domain is presented in Section VI.

B.5 Workflow Management and Cooperation

Workflow management applications support the cooperation of persons and tools involved in an engineering or business process. The workflow defines which activities must be carried out to accomplish a given task as well as how, where, and when these activities involve each party. A way to model this is to represent activities as autonomous entities that, during their evolution, are circulated among the entities involved in the workflow. Code mobility could be used to provide support for mobility of activities that encapsulate their definition and state. For example, a mobile component could encapsulate a text document that undergoes several revisions. The component maintains information about the document state, the legal operations on its contents, and the next scheduled step in the revision process. An application of these concepts can be found in [59].

B.6 Active Networks

The idea of active networks has been proposed recently [60], [61] as a means to introduce flexibility into networks and provide more powerful mechanisms to "program" the network according to applications' needs. Although some interpret the idea of active networks without any relation with code mobility [62], most of the approaches rely on it. They can be classified along a spectrum delimited by two extremes represented by the programmable switch and the capsule approaches [60]. The *programmable switch* approach is basically an instantiation of the COD paradigm, and aims at providing dynamic extensibility of network devices through dynamic linking of code. On the other hand, the *capsule* approach proposes to attach to every packet flowing in the network some code describing a computation that must be performed on packet data, at each node. Clearly, active networks aim at leveraging off of the advantages provided by code mobility in terms of deployment and maintenance, customization of services, and protocol encapsulation. As an example, in this scenario a multiprotocol router could download on demand the code needed to handle a packet corresponding to an unknown protocol, or even receive the protocol together with the packet. The work described in [63] is an example of an active network architecture exploiting the COD paradigm.

B.7 Electronic Commerce

Electronic commerce applications enable users to perform business transactions through the network. The application environment is composed of several independent and possibly competing business entities. A transaction may involve negotiation with remote entities and may require access to information that is continuously evolving, e.g., stock exchange quotations. In this context, there is the need to customize the behavior of the parties involved in order to match a particular negotiation protocol. Moreover, it is desirable to move application components close to the information relevant to the transaction. This problems make mobile code appealing for this kind of applications. Actually, Telescript [64] was conceived explicitly to support electronic commerce. For this reason, the term "mobile agent" is often related with electronic commerce. Another application of code mobility to electronic commerce can be found in [65].

VI. A Case Study in Network Management

The purpose of this section is to illustrate how the classification we presented so far can be used to guide the software engineer through the design and implementation phases of the application development process. To this end, we focus on the typical functionality required to a network management application, i.e., the polling of management information from a pool of network devices. Current protocols are based on a centralized client-server paradigm that exhibits several drawbacks [57], discussed in Section VI-A. The identification and evaluation of alternative solutions will be discussed in the remainder of this section.

The suggested development process proceeds as follows. Given an application whose requirements have been already specified, the first step is to determine if the mobile code approach is suited to meet the application needs—that is, whether we have to use code mobility at all. This early evaluation is performed on the basis of the discussion at the beginning of Section V. The second step involves identifying the suitable paradigms for the design of the application at hand. This is done informally and qualitatively, as in the case described in Section VI-B. Then, the tradeoffs among the various paradigms must be analyzed *for each application functionality* whose design could involve code mobility. To achieve this, in Section VI-C we build a model of the application functionality that enables quantitative analysis of the tradeoffs, along the lines of [4]. Finally, after the suitable paradigms have been chosen, the technology for implementation has to be selected by examining the tradeoffs highlighted in Section IV, e.g., trading ease of programming for lightweight implementation. This will be discussed in Section VI-D.

We chose network management as the application domain for our case study because, although it is often indicated as the ideal testbed for code mobility, efforts in this direction are still in their early stages [58], [60]. The results illustrated in the remainder of this section represent the preliminary achievements of on-going work on the subject [50], [66].

A. The Problem: Decentralizing Network Traffic

The world of network management research can be split roughly in two worlds: management of IP networks, where the *Simple Network Management Protocol* [67] proposed by IETF is the dominant protocol, and management of ISO networks, based on the *Common Management Information Protocol* [68]. Both protocols are based on a CS paradigm where a network management station—the client—polls information from *agents*[5]—the servers—residing on the network devices. Each agent is in charge of managing a *management information base* (MIB)[6], a hierarchical base of information that stores the relevant parameters of the corresponding device. In this setting, all the computation related to management, e.g., statistics, is demanded to the management station. Polling is performed using very low level primitives—basically `get` and `set` of atomic values in the MIB. This fine grained CS interaction is often called *micro management*, and leads to the generation of intense traffic and computational overload on the management station. This centralized architecture is particularly inefficient during periods of heavy congestion, when management becomes important. In fact, during these periods the management station increases its interactions with the devices and possibly uploads configuration changes, thus increasing congestion. In turn, congestion, as an abnormal status, is likely to trigger notifications to the management station which worsen network overload. Due to this situation, access to devices in the congested area becomes difficult and slow.

[5] Despite the name, management agents are conventional programs which cannot move and in general do not exhibit a great deal of intelligence.

[6] MIB is actually the term used for information bases in SNMP only. CMIP uses the term *management information tree* (MIT) database instead. Hereinafter, we will ignore the difference for the sake of simplicity.

These problems have been addressed by IETF and ISO with modifications of their management architecture. For instance, SNMPv2 [69] introduced hierarchical decentralization through the concept of *proxy agents*. A proxy agent is responsible for the management of a pool of devices (towards which it acts as a client) on behalf of the network management station (towards which it acts as a server). Another protocol derived from SNMP, called Remote MONitoring (RMON) [70], assumes the existence of stand-alone dedicated devices called *probes*. Each probe hosts an agent able to monitor "global" information flowing through links rather than information "local" to a device. Although these decentralization features improve the situation, experimentation showed that they do not provide the desired level of decentralization needed to cope with large networks.

As discussed in Section V, network management applications may overcome some of these limits by taking advantage of the benefits of the mobile code approach, such as dynamicity in service deployment and customization, autonomy, and fault tolerance.

B. Identifying the Design Paradigms

In this section, we analyze if and how the mobile code design paradigms described in Section IV can provide a suitable alternative to the CS paradigm fostered by SNMP, and thus help in solving the problems depicted above.

The rationale for the management architecture proposed in SNMP and CMIP, which provides very low-granularity primitives, is to keep the agents on the devices small and easily implementable, keeping all the complexity on the management station. Nevertheless, as we described earlier, this is going to dramatically increase congestion and decrease performance. For instance, tables are often used to store information into devices. To search a value in a table using a CS approach, either the table has to be transferred to the management station and searched there for the desired value, or the agent has to be modified to provide a new search service. Neither solution is desirable. The former leads to bandwidth waste for large tables. The second increases the size of the agent as a larger number of routines are implemented—maybe without a substantial payoff if the routines are used only now and then.

The REV paradigm could be used to pack together the set of SNMP operations describing the search and send them on the device holding the table for local interaction[7]. After execution, only the target value should be sent back—thus performing semantic compression of data. Intuitively, this solution is likely to save bandwidth at least for big tables and small routines. As an aside, this solution provides a desirable side-effect: it raises the level of abstraction of the operations available to the network manager. One could envision a scenario where the manager builds her own management procedures upon lower level primitives, stores them on the management station, and invokes their remote evaluation on the appropriate device whenever needed.

[7]We assume the presence of a run-time support for mobile code on network devices. This assumption could be considered unrealistic only a couple of years ago. Today, some device manufacturers already announced support for Java in the next releases of their systems.

On the other hand, the capability to retain the state across several hops implicit in an MA design adds a new dimension to the benefits achievable through an REV design: autonomy. In the REV paradigm each remote evaluation on a device must be initiated explicitly by the management station. In the MA paradigm, the management station can exploit the capability of a mobile component to retain its state and demand to it the retrieval of information from a specified pool of devices. Thus, it can delegate to it the decision about when and where to migrate, according to its current state. Whether this is actually improving traffic load is still unclear at this point, because the state of the mobile component is likely to grow from hop to hop. This issue will be analyzed later. Nevertheless, some other advantages which can determine the choice of the MA paradigm independently of the issue of traffic are worth to be mentioned. For instance, let us consider a scenario where the pool of devices to be managed resides in a LAN, and assume that the management station is connected to the managed devices by a long-haul link, likely to be unreliable and slow. In this case the mobile component, once injected into the LAN, can collect information about all the managed devices without any need to be connected with the management station. Even if the state of the mobile component increases during this operation, bandwidth is assumed to be cheaper within the LAN than on the long-haul link. In addition, mobile components could have the capability to operate even when network level routing is disrupted. If the management station does not have network level connectivity with a node to be managed, it can provide its mobile component with a route calculated from historical routing information and send it to the first hop on the route. Whenever the mobile component resumes execution on an intermediate hop, it tries to reach one of the next hop towards the target node using its internal route, until it reaches the target and performs the management task.

The COD paradigm gives only a partial solution to the problem, as it provides the capability to extend dynamically the set of services offered by a device. This is convenient if many identical queries have to be performed on a device: once the code to perform the SNMP queries locally is installed, it can be remotely invoked by the management station. On the other hand, if few different queries have to be performed, COD does not help that much: either a REV or MA paradigm need to be exploited. In the following, we will focus our discussion only on these two paradigms.

C. Evaluating the Design Tradeoffs

The previous section has emphasized from an informal and qualitative viewpoint several advantages of mobile code design paradigms over the traditional CS paradigm. Nevertheless, as we pointed out in Section IV, mobile code design paradigms are not good per se. Rather their application must be carefully analyzed on a case-by-case basis, taking into account traditional paradigms as well. In this section, we exemplify this concept by comparing formally and quantitatively different solutions to the problem of polling device data with respect to the traffic they generate.

The scenario we assume is the following. A network management station retrieves management data from a pool of devices, e.g., the load on every network interface of each device. Data retrieval is conceptually a single query on the device, but is actually implemented by several SNMP instructions. Table III shows a set of parameters needed to model this scenario. Such parameters define an oversimplified model. For instance, CPU time is considered an infinite resource and, even more important, the network is considered uniform, with no difference in bandwidth or latency among the links—a heavy assumption for network management. Finally, in real protocols like TCP the overhead h actually depends on the payload size. Nevertheless, our goal here is to illustrate some guidelines to evaluate the tradeoffs among paradigms: a quantitative comparison among paradigms, encompassing a precise characterization of network management functionalities and an accurate model of network protocols, can be found in [66].

A design exploiting the CS paradigm fostered by SNMP would lead to an overall traffic described by the expression

$$T_{CS} = (2h + i + r)QN.$$

In fact, due to the SNMP architecture, each of the Q instructions implementing the query has to be sent separately on each of the N nodes and returns a single result r which is collected and subsequently elaborated by the management station.

Exploitation of the REV paradigm assumes that the set of Q SNMP instructions representing the query are embedded in mobile code sent to each device and executed remotely. If the management station is interested in all the results returned by each SNMP instruction (which are shipped altogether) and we assume a value C_{REV} for the size of the code sent, the expression for the traffic is

$$T_{REV} = (2h + C_{REV} + rQ)N.$$

Finally, in a design based on the MA paradigm the code encapsulating the query can move autonomously among the network devices retaining its state, which is growing as long as the mobile component collects information. The expression for the overall traffic, assuming a value C_{MA} for the size of the mobile component, is then

$$T_{MA} = (h + C_{MA})(N+1) + \frac{rQN(N+1)}{2}.$$

This analysis shows that the MA paradigm is never convenient, at least as far as overall network traffic is concerned. On the other hand, assuming that $2h \ll C_{REV}$, REV is more convenient than CS if

$$\frac{C_{REV}}{Q} < (2h + i)$$

holds, where variables in the right hand side depend on the SNMP protocol and those in the left hand side depend on the particular network configuration and functionality. The formula above proves the intuition that REV is convenient when a set of SNMP instructions can be "packed" efficiently into mobile code, e.g., by exploiting loops. Nevertheless, the formula gives a quantification about when to use a paradigm rather than the other.

Although overall traffic is an important parameter to optimize, we pointed out earlier that one of the key benefits of mobile code is that it enables decentralized network management, reducing the load on the management station. With the CS and REV paradigms, the expression for the traffic around the management workstation coincides with the expression for the overall traffic. Instead, an MA design involves the management station only when the mobile component is injected into the network and when it comes back to the station, giving the expression

$$T_{MA}^{Mgm} = 2(h + C_{MA}) + rQN.$$

In other words, the traffic around the workstation is diminished, that is $\Delta T = T_{CS} - T_{MA}^{Mgm} > 0$, when

$$\frac{C_{MA}}{QN} < \frac{(2h+i)}{2},$$

assuming $QN \gg 1$. Again, this provides quantitative evidence for the fact that improvement of traffic increases with the number of nodes being managed autonomously by the mobile component and with the number of instructions that can be packed efficiently into the component code.

It is worth noting that small changes in the model can modify slightly the tradeoff. For example, if semantic compression of data is performed, e.g., because the management station is interested only in the maximum among the Q values retrieved on each the device, the expression for the traffic in the MA case becomes

$$T'_{MA} = (h + C_{MA})(N+1)\frac{rN(N+1)}{2},$$

and can even become linear, that is

$$T''_{MA} = (h + C_{MA})(N+1) + rN,$$

when semantic compression can be performed across all the devices (e.g., because the management station is interested in finding the maximum value among all the devices). This would make MA a candidate even in absence of congestion around the management station.

The analysis just carried out evidences that, as far as code mobility is concerned, REV and MA are the design paradigms one may want to exploit in designing a polling functionality for a network management application. Nevertheless, if the actual values of the application parameters are in a certain range, it still desirable to use a CS paradigm. Hence, the choice of the paradigm is constrained by the actual values for the parameters of the application.

As a final remark, it should be pointed out that, although network traffic is a key parameter in the context

parameter	unit	description
N	node	number of managed network devices
Q	instruction	number of SNMP instructions needed to perform a single device query
i	bit	size of an SNMP instruction
h	bit	message header and other auxiliary data encapsulating message content
r	bit	average size of the result of an SNMP instruction

TABLE III

Parameters modeling a simple network management data retrieval functionality.

of network management, in other applications it might be completely irrelevant and other factors may be predominant, e.g., CPU usage. In these cases, the same approach based on quantitative analysis can be put in place.

D. Choosing the Implementation Technology

In principle, design paradigms and the technology used for their implementation are orthogonal, as discussed at the end of Section IV. Nevertheless, we have already pointed out that this is true only partially, and that an inappropriate technology may put an unnecessary burden on the programmer—at least as far as code mobility is concerned. In particular, we showed how a strong MCS is the natural choice for implementing an MA design. Mobility is reduced to a single instruction, and the migrating EU can be mapped directly to a roaming component in the higher-level design. Conversely, a weak MCS constrains the programmer to manage explicitly the execution state, which degradates programmer productivity, program readability, and ease of debugging. In the context of our case study, however, there is an additional drawback. The formulas we derived in the previous section show how the size of the transferred code is a key parameter in the expressions of network traffic. Implementing an MA design with a weak MCS is likely to end up in creating bigger code (because of the explicit management of execution state), thus reducing the benefits potentially achievable.

Nevertheless, the final choice might be influenced by other considerations as well. The analysis described in the previous section aims at identifying the best paradigm to design a single functionality within an application. Of course, an application is composed of several functionalities, each with its own peculiarities that may lead to completely different designs. For instance, suppose we want to implement a network management application that provides, among the others, a first functionality to determine the most loaded interface on a given path, a second functionality that determines all the parameters for a given interface, and a third one that allows the manager to set a given value in a device's MIB. The analysis carried out earlier tells us that in the first case we may want to take advantage of the opportunity to perform global semantic compression, and exploit the MA paradigm; in the second, we may want to follow the REV paradigm to save bandwidth—MA is overshooting. In the third one, CS will suffice.

The choice of the technology used to implement the application must take into account not only which is the best technology to implement a given functionality designed following a certain design paradigm, but also how the technology fits the global application development. For example, let us suppose that we are faced with the choice between a strong MCS that does not provide good support for standalone code shipping and a weak MCS that provides it. In the context depicted above, the first functionality is likely to be used less frequently than the others: in this case, we may want to sacrifice the traffic optimization achievable with the strong MCS and use the weak one, to obtain better support in the key functionalities and keep the uniformity of the development tools.

VII. Conclusions

Mobile code is a promising solution for the design and implementation of large scale distributed applications, since it overcomes many of the drawbacks of the traditional client-server approach. However, most research efforts in this field have been focused on the development of mobile code technologies, and little attention has been payed so far to the formulation of a sound conceptual framework for code mobility.

In this paper we proposed a conceptual framework structured along three classes of concepts: *applications*, *design paradigms*, and *technologies*. Applications are the solutions to specific problems. Paradigms guide the design of applications. Technologies support application development. We surveyed each of these concepts and pointed out features, advantages, and disadvantages of existing approaches and proposals. The purpose of the framework presented in this work is to foster progress towards a common understanding of the issues and contributions in the area of code mobility. The framework will be a useful guideline to practitioners, who can use it to exploit the potential of the different mobile code concepts and technologies.

Certainly, the work presented in this paper needs to be incrementally enriched and revised, taking into account experiences, results, and innovations as they emerge from the research activity. In particular, we need to improve our understanding of the properties and weaknesses of the existing design paradigms. We also need to consolidate a detailed conceptual framework for mobile code languages, that makes it possible to compare them as we do for traditional programming languages [71]. Another issue is the development of models enabling formal reasoning and verification. Finally, we need to further explore the relatively unknown world of applications and problems that can benefit from the adoption of technology and methodology based on mobile code. Nonetheless, we believe that the concepts presented in this paper can be instrumental in the creation of a mature and comprehensive background for the evolution and further diffusion of mobile code applications and techniques.

Acknowledgments

We wish to thank Mario Baldi, Antonio Carzaniga, Gianpaolo Cugola, and Carlo Ghezzi. Without the lively discussions with them and the related work developed jointly, this paper would have never been written.

References

[1] G.H. Forman and J. Zahorjan, "The Challenges of Mobile Computing," *IEEE Computer*, vol. 27, no. 4, pp. 38–47, 1994.

[2] T. Imielinsky and B.R. Badrinath, "Wireless Computing: Challenges in Data Management," *Comm. of the ACM*, vol. 37, no. 10, pp. 18–28, 1994.

[3] Object Management Group, *CORBA: Architecture and Specification*, Aug. 1995.

[4] A. Carzaniga, G.P. Picco, and G. Vigna, "Designing Distributed Applications with Mobile Code Paradigms," in *Proc. of the 19^th^ Int. Conf. on Software Engineering (ICSE'97)*, R. Taylor, Ed. 1997, pp. 22–32, ACM Press.

[5] J.K. Boggs, "IBM Remote Job Entry Facility: Generalize Subsystem Remote Job Entry Facility," IBM Technical Disclosure Bulletin 752, IBM, Aug. 1973.

[6] Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison-Wesley, 1985.

[7] M. Nuttall, "Survey of systems providing process or object migration," Tech. Rep. Doc 94/10, Dept. of Computing, Imperial College, May 1994.

[8] G. Thiel, "Locus operating system, a transparent system," *Computer Communications*, vol. 14, no. 6, pp. 336–346, 1991.

[9] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained Mobility in the Emerald System," *ACM Trans. on Computer Systems*, vol. 6, no. 2, pp. 109–133, Feb. 1988.

[10] R. Lea, C. Jacquemont, and E. Pillevesse, "COOL: System Support for Distributed Object-Oriented Programming," *Comm. of the ACM*, vol. 36, no. 9, pp. 37–46, Nov. 1993.

[11] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, P. Leonard, S. Langlois, and W. Neuhauser, "Chorus Distributed Operating Systems," *Computing Systems*, vol. 1, pp. 305–379, Oct. 1988.

[12] C.G. Harrison, D.M. Chess, and A. Kershenbaum, "Mobile Agents: Are they a good idea?," In Vitek and Tschudin [73], pp. 25–47, Also available as IBM Technical Report.

[13] L. Bic, M. Fukuda, and M. Dillencourt, "Distributed Computing Using Autonomous Objects," *IEEE Computer*, Aug. 1996.

[14] J.W. Stamos and D.K. Gifford, "Implementing Remote Evaluation," *IEEE Trans. on Software Engineering*, vol. 16, no. 7, pp. 710–722, July 1990.

[15] A. Birrell and B. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems*, vol. 2, no. 1, pp. 29–59, Feb. 1984.

[16] E.J.H. Chang, "Echo Algorithms: Depth Parallel Operations on General Graphs," *IEEE Trans. on Software Engineering*, July 1982.

[17] D. Johansen, R. van Renesse, and F.B. Schneider, "An Introduction to the TACOMA Distributed System - Version 1.0," Tech. Rep. 95-23, Dept. of Computer Science, Univ. of Tromsø and Cornell Univ., Tromsø, Norway, June 1995.

[18] A.S. Park and S. Leuker, "A Multi-Agent Architecture Supporting Services Access," In Rothermel and Popescu-Zeletin [72], pp. 62–73.

[19] J.E. White, "Telescript Technology: Mobile Agents," in *Software Agents*, J. Bradshaw, Ed. AAAI Press/MIT Press, 1996.

[20] P. Maes, "Agents that Reduce Work and Information Overload," *Comm. of the ACM*, vol. 37, no. 7, July 1994.

[21] M. Genesereth and S. Ketchpel, "Software Agents," *Comm. of the ACM*, vol. 37, no. 7, July 1994.

[22] M. Wooldridge and N.R. Jennings, "Intelligent Agents: Theory and Practice," *Knowledge Engineering Review*, vol. 10, no. 2, June 1995.

[23] F.C. Knabe, *Language Support for Mobile Agents*, Ph.D. thesis, Carnegie Mellon Univ., Pittsburgh, PA, USA, Dec. 1995, Also available as Carnegie Mellon School of Computer Science Technical Report CMU-CS-95-223 and European Computer Industry Centre Technical Report ECRC-95-36.

[24] H. Peine and T. Stolpmann, "The Architecture of the Ara Platform for Mobile Agents," In Rothermel and Popescu-Zeletin [72], pp. 50–61.

[25] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet, "Concordia: An Infrastructure for Collaborating Mobile Agents," In Rothermel and Popescu-Zeletin [72], pp. 86–97.

[26] M. Fukuda, L. Bic, M. Dillencourt, and F. Merchant, "Intra-Inter-Object Coordination with MESSENGERS," in *1^st^ Int. Conf. on Coordination Models and Languages (COORDINATION'96)*, 1996.

[27] J. Kiniry and D. Zimmerman, "A Hands-On Look at Java Mobile Agents," *IEEE Internet Computing*, vol. 1, no. 4, pp. 21–30, 1997.

[28] D. Volpano, "Provably-Secure Programming Languages for Remote Evaluation," *ACM Computing Surveys*, vol. 28A, Dec. 1996, Participation statement for ACM Workshop on Strategic Directions in Computing Research.

[29] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna, "Analyzing Mobile Code Languages," In Vitek and Tschudin [73], pp. 93–111.

[30] R.S. Gray, "Agent Tcl: A transportable agent system," in *Proc. of the CIKM Workshop on Intelligent Information Agents*, Baltimore, Md., Dec. 1995.

[31] B. Thomsen, L. Leth, S. Prasad, T.-M. Kuo, A. Kramer, F.C. Knabe, and A. Giacalone, "Facile Antigua Release programming guide," Tech. Rep. ECRC-93-20, European Computer Industry Research Centre, Munich, Germany, Dec. 1993.

[32] Sun Microsystems, "The Java Language: An Overview," Tech. Rep., Sun Microsystems, 1994.

[33] D.B. Lange, "Java Aglets Application Programming Interface (J-AAPI)," IBM Corp. White Paper, Feb. 1997.

[34] D.B. Lange and D.T. Chang, "IBM Aglets Workbench—Programming Mobile Agents in Java," IBM Corp. White Paper, Sept. 1996.

[35] C. Tschudin, *An Introduction to the* `MO` *Messenger Language*, Univ. of Geneva, Switzerland, 1994.

[36] C. Tschudin, "OO-Agents and Messengers," in *ECOOP'95 Workshop W10 on Objects and Agents*, Aug. 1995.

[37] M. Straßer, J. Baumann, and F. Hohl, "Mole—A Java Based Mobile Agent System," in *Special Issues in Object-Oriented Programming: Workshop Reader of the 10^th^ European Conf. on Object-Oriented Programming ECOOP'96*, M. Mühlaüser, Ed. July 1996, pp. 327–334, dpunkt.

[38] J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, and M. Straßer, "Communication Concepts for Mobile Agent Systems," In Rothermel and Popescu-Zeletin [72], pp. 123–135.

[39] J. Hogg, "Island: Aliasing Protection in Object-Oriented Languages," in *Proc. of OOPSLA '91*, 1991.

[40] L. Cardelli, "A language with distributed scope," *Computing Systems*, vol. 8, no. 1, pp. 27–59, 1995.

[41] N. Borenstein, "EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail," Tech. Rep., First Virtual Holdings, Inc, 1994.

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

[43] J. Ousterhout, J. Levy, and B. Welch, "The Safe-Tcl Security Model," Tech. Rep., Sun Microsystems, Nov. 1996, Reprinted in [52].

[44] A. Acharya, M. Ranganathan, and J. Saltz, "Sumatra: A Language for Resource-aware Mobile Programs," In Vitek and Tschudin [73], pp. 111–130.

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

[46] G. Abowd, R. Allen, and D. Garlan, "Using Style to Understand Descriptions of Software Architecture," in *Proc. of SIGSOFT'93: Foundations of Software Engineering*, Dec. 1993.

[47] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A Note on Distributed Computing," In Vitek and Tschudin [73], Also available as Sun Microsystems Laboratories Technical Report TR-94-29.

[48] G.P. Picco, G.-C. Roman, and P.J. McCann, "Expressing Code Mobility in Mobile UNITY," in *Proc. of the 6^{th} European Software Engineering Conf. held jointly with the 5^{th} ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE '97)*, M. Jazayeri and H. Schauer, Eds., Zurich, Switzerland, Sept. 1997, vol. 1301 of *LNCS*, pp. 500–518, Springer.

[49] C. Ghezzi and G. Vigna, "Mobile Code Paradigms and Technologies: A Case Study," In Rothermel and Popescu-Zeletin [72], pp. 39–49.

[50] M. Baldi, S. Gai, and G.P. Picco, "Exploiting Code Mobility in Decentralized and Flexible Network Management," In Rothermel and Popescu-Zeletin [72], pp. 13–26.

[51] K.A. Bharat and L. Cardelli, "Migratory Applications," Tech. Rep. 138, Digital Equipment Corporation, Systems Research Center, Feb. 1996.

[52] G. Vigna, Ed., *Mobile Agents and Security*, LNCS State-of-the-Art Survey. Springer, 1998.

[53] P. Knudsen, "Comparing two Distributed Computing Paradigms - a Performance Case Study," M.S. thesis, Univ. of Tromsø, 1995.

[54] A. Limongiello, R. Melen, M. Roccuzzo, A. Scalisi, V. Trecordi, and J. Wojtowicz, "ORCHESTRA: An Experimental Agent-based Service Control Architecture For Broadband Multimedia Networks," GLOBAL Internet '96, Nov. 1996.

[55] T. Magedanz, K. Rothermel, and S. Krause, "Intelligent Agents: An Emerging Technology for Next Generation Telecommunications?," in *INFOCOM'96*, San Francisco, CA, USA, Mar. 1996.

[56] R.S. Gray, D. Kotz, S. Nog, D. Rus, and G. Cybenko, "Mobile agents for mobile computing," in *Proc. of the 2^{nd} Aizu Int. Symp. on Parallel Algorithms/Architectures Synthesis*, Fukushima, Japan, Mar. 1997.

[57] Y. Yemini, "The OSI Network Management Model," *IEEE Communications*, pp. 20–29, May 1993.

[58] G. Goldszmidt and Y. Yemini, "Distributed Management by Delegation," in *Proc. of the 15^{th} Int. Conf. on Distributed Computing*, June 1995.

[59] T. Cai, P. Gloor, and S. Nog, "DataFlow: A Workflow Management System on the Web using transportable Agents," Tech. Rep. TR96-283, Dept. of Computer Science, Dartmouth College, Hanover, NH, 1996.

[60] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden, "A Survey of Active Network Research," *IEEE Communications*, vol. 35, no. 1, pp. 80–86, Jan. 1997.

[61] Y. Yemini and S. da Silva, "Towards Programmable Networks," in *IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, Oct. 1996.

[62] S. Bhattacharjee, K.L. Calvert, and E.W. Zegura, "An Architecture for Active Networking," in *High Performance Networking (HPN'97)*, Apr. 1997.

[63] D.J. Wetherall, J. Guttag, and D.L. Tennenhouse, "ANTS: A Toolkit for Building an Dynamically Deploying Network Protocols," Tech. Rep., MIT, 1997, Submitted for publication to IEEE OPENARCH'98.

[64] J.E. White, "Telescript Technology: The Foundation for the Electronic Marketplace," Tech. Rep., General Magic, Inc., 1994, White Paper.

[65] M. Merz and W. Lamersdorf, "Agents, Services, and Electronic Markets: How do they Integrate?," in *Proc. of the Int'l Conf. on Distributed Platforms*. IFIP/IEEE, 1996.

[66] M. Baldi and G.P. Picco, "Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications," in *Proc. of the 20^{th} Int. Conf. on Software Engineering*, R. Kemmerer, Ed., 1998, To appear.

[67] J.D. Case, M. Fedor, M. L. Schoffstall, and C. Davin, "Simple Network Management Protocol," RFC 1157, May 1990.

[68] OSI, "ISO 9595 Information Technology, Open System Interconnection, Common Management Information Protocol Specification," 1991.

[69] J.D. Case, K. McCloghrie, M. Rose, and S. Waldbusser, "Structure of Management Information for version 2 of the Simple Network Management Protocol," RFC 1902, Jan. 1996.

[70] S. Waldbusser, "Remote Network Monitoring Management Information Base," RFC 1757, Feb. 1995.

[71] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, John Wiley & Sons, 3^{rd} edition, 1997.

[72] K. Rothermel and R. Popescu-Zeletin, Eds., *Mobile Agents: 1^{st} International Workshop MA '97*, vol. 1219 of *LNCS*. Springer, Apr. 1997.

[73] J. Vitek and C. Tschudin, Eds., *Mobile Object Systems: Towards the Programmable Internet*, vol. 1222 of *LNCS*, Springer, Apr. 1997.

Alfonso Fuggetta is Associate Professor of Software Engineering at Politecnico di Milano. He is also Senior Researcher at CEFRIEL, a research and education institute established in Milano by universities, the regional council of Lombardy, and several major IT industries. His research interest are in workflow and process modeling and support, technologies and methods for distributed and mobile systems, requirement engineering. He is member of IEEE, IEEE Computer Society, and ACM. More info can be found at `http://www.elet.polimi.it/~fuggetta`.

Gian Pietro Picco holds a Dr.Eng. degree in electronic engineering from Politecnico di Milano, Italy, and a Ph.D. degree in computer engineering from Politecnico di Torino, Italy. The subject of his recent Ph.D. dissertation and of his current research is understanding, evaluating, formalizing, and exploiting code mobility in the context of large-scale distributed systems. Prior to that, he published work in software process modeling, object-oriented databases, and robotics. He is presently a visiting researcher at Washington University, St. Louis, USA, where he is investigating the relationships between mobile code and mobile computing. He is member of IEEE, IEEE Computer Society, and ACM. More info can be found at `http://www.polito.it/~picco`.

Giovanni Vigna received the Dr.Eng. degree in electronic engineering in 1994 and the Ph.D. degree in computer engineering 1998 from Politecnico di Milano, Italy. His Ph.D. dissertation focused on mobile code technologies and design paradigms, with an emphasis on security issues. He authored several publications on mobile code and he is editor of a special issue of the LNCS on mobile code and security. He is currently with University of California, Santa Barbara, as a post doc researcher. His research interests include mobile code, WWW engineering, electronic commerce, network security, and intrusion detection. He is a member of IEEE, IEEE Computer Society, and ACM. More info can be found at `http://www.elet.polimi.it/~vigna`.