POLITECNICO DI MILANO
Dottorato di Ricerca in Ingegneria Informatica e Automatica

Mobile Code Technologies, Paradigms, and Applications

Tesi di Dottorato di:
Giovanni Vigna

Relatore:
Prof. Carlo Ghezzi
Correlatore:
Prof. Rocco De Nicola
Tutore:
Prof. Alfonso Fuggetta
Coordinatore del dottorato:
Prof. Sergio Bittanti

X ciclo

La donna è mobile qual piuma al vento...

Acknowledgements

I want to thank Prof. Carlo Ghezzi that, some years ago, suggested the theme of this research and, since then, followed with constant attention my work, providing ideas, guidelines, and suggestions. In particular, I am grateful for the support he gave me in all the difficult moments of these years. I want to thank Prof. Alfonso Fuggetta that introduced me to the PhD and tutored me during these years, providing constant support. I want to thank the tutors of my minors, Prof. Maurizio Decina and Prof. Franca Garzotto.

I cannot forget all the people that shared the PhD experience and the research work: Leonardo Bonomi, Francesco Calzolari, Antonio Carzaniga, Fabiano Cattaneo, Francesco Coda, Gianpaolo Cugola, Elisabetta Di Nitto, Alessandro Orso, Gian Pietro Picco, Sergio Silva, and many others.

I want to thank Prof. Richard Kemmerer that gave me the opportunity to continue my research at the University of Califonia in Santa Barbara, and many firends at UCSB, that helped me in many different ways: Giulia Brofferio, Diana Hadjes, Joanne Hollyday, Andrea Fiore, Matt Lauer, Luca Lucchese, Maria Teresa Napoli, Nicola Orio, Regina Sherman, Sharon Webb, and many many others.

A particular thank goes to my wife, Luisa, that supported me in these years. Finally, many thanks to my parents, Carmelo and Mirella, and to my brother Sebastiano that probably participated to my PhD more than myself.

Contents

Chapter 1

Introduction

1.1 Moving Code

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 increased traffic, which in turn triggers significant efforts to enhance the *performance* of the communication infrastructure. Network links are constantly improved, and technological developments lead to increased computational power on both intermediate and end network nodes.

The increase in size and performance of computer networks is both the cause and the effect of an important phenomenon: networks are becoming pervasive and ubiquitous. By *pervasive*, we mean that network connectivity is no longer an expensive add-on, rather it is often implicitly assumed when considering a computing platform. By *ubiquitous*, we refer to the availability of network connectivity independently of the physical location. Developments in wireless technology free network hosts from the constraint of being placed at a fixed physical location and enable a new scenario where hosts, still connected to the net by wireless links, follow their users while their physical location is changing. This approach, often referred to as *mobile computing* or *nomadic computing*, is gaining popularity thanks to the diffusion of laptops and personal digital assistants (PDAs).

At the same time, the availability of distributed applications and systems accessible to the general public (e.g., the World Wide Web) has triggered the interest of end users and markets, motivating the development of entirely new classes of applications. This is changing the way networks, and particularly the Internet, are exploited. Their *role* is shifting progressively from that of plain communication media to large scale distributed platforms that support innovative and previously unforeseen services. Terms like "electronic commerce", or "Internet telephony" are symptomatic of this shift.

Nonetheless, the range of applications capable of fully exploiting the underlying communication infrastructure is still relatively limited. This is mainly due to the lack of suitable mechanisms and abstractions that would allow developers to take full advantage of the potential of the new communication infrastructure. In other words, the *computational infrastructure*, i.e., the software layer that provides the set of primitives and mechanisms upon which distributed applications are built, is not keeping the pace with the evolution of the communication infrastructure. There are several issues to 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, whose exploitation in computer networks is often referred to as *mobile computing*, poses even tougher problems [28, 43]. 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 adaptation of well-known theoretical and technological results is needed. On the other hand, the new services and modes of use of computer networks often demand for a higher degree of *customizability* by the end user, whereas the dynamic nature of both the underlying communication infrastructure and the market requirements demands for increased *flexibility* and *extensibility*.

There have been many attempts to design a computational infrastructure that can match the characteristics of the communication infrastructure. These efforts attack different facets of the problem at different layers of abstraction. Most of the approaches, however, try to adapt well-established models and technology within the new setting, and usually take for granted the traditional client-server architecture. For example, CORBA [66] 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, extending existing client-server technology is only a partial solution because it addresses just scaling and connectivity problems without providing new mechanisms and abstractions that allow the developers to structure the computation in a flexible, customizable, and extensible way.

A different approach comes from a promising research area which is attacking the problems mentioned so far by exploiting *mobile code*. Code mobility can be defined informally as the capability to change dynamically the bindings between code fragments and the location where they are executed [18]. This powerful concept 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 common framework is still missing: terms are defined imprecisely and with a loose semantics. Similar terms are used to denote completely different entities (a notable example is the term *agent*) and similar entities are referenced using different names, leading to confusion[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

[1]There is not even a commonly agreed term to qualify the subject of this research 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*.

sometimes unjustified expectations.

The market interest in mobile code technologies and the fast evolution of the Internet infrastructure is motivating research efforts focused on timely deliver of prototypes rather than on understanding the issues raised by code mobility and on the conscious design of new constructs and abstractions that may help in implementing mobile code applications. Since there is seldom an explicit declaration of the scope of the technology, functionalities and potentials are hidden. This makes it difficult to compare and assess the characteristics of the different solutions. In addition, security, with few notable exceptions, is not introduced since from the first phases of the technology development, and it is left as "future work". Since moving code across a network worsens well-known security problems and raises completely new security issues, this approach is hampering the wide acceptance of the mobile code approach.

1.2 Contribution of this thesis

This thesis addresses the problems described above at the methodological and technological level.

First, *we introduce a framework of concepts, models, and terms to understand, assess and compare the different facets of the mobile code approach.*

In particular:

- we develop a taxonomy of mobile code technologies, classifying different types of mobility and several alternative mechanisms for communication, execution, and security;
- we apply our taxonomy to several existing technologies, highlighting their distinguishing characteristics and enabling assessment and comparison;
- we define and introduce mobile code paradigms—architectural styles that model, at the design level, the interaction patterns defining the relocation of the components of a distributed software architecture exploiting code mobility;
- we analyze and classify a number of application domains that may benefit from the use of the mobile code approach.

Second, *we use the taxonomy to develop guidelines for the selection of design paradigms and technologies during the development process of a mobile code application. By using a case study we highlight the scope of the different concepts, their mutual relationships, and how developers can take advantage of the proposed framework.*

Third, *we introduce a novel mechanism based on cryptographic techniques that solves the problem of protecting mobile code from hostile execution environments. This problem was previously considered unsolvable.*

1.3 Structure of this thesis

This thesis is structured as follows. Chapter 2 presents some related work to understand the essence and the potential of code mobility, and introduces the structure of our framework. Chapters from 3 to 5 develop the details of the framework. Chapter 6 presents a case study that describes how the framework supports the developer of mobile code applications. Chapter 7 describes a new mechanism, called *cryptographic tracing* that allows users to protect mobile programs from attacks coming from malicious sites. Finally, Chapter 8 summarizes the contribution of this thesis and outlines directions for future research in the field.

Chapter 2

Motivation and approach

2.1 Related work

Code mobility is not a new concept. In the recent past, several mechanisms and facilities have been designed and implemented to support the migration of code among the nodes of a network. Examples are remote batch job submission [12] and the use of PostScript [44] to control printers. A more structured approach has been followed in the research work on distributed operating systems. In this context, the main problem is to support the migration of active processes and objects (together with their state and the associated code) at the operating system level. A survey of systems providing process or object migration can be found in [65].

Process migration concerns the transfer of an operating system process from the machine where it is running to a different one. Migration mechanisms manage the bindings between the process and its execution environment (e.g., open file descriptors and environment variables) to allow the process to resume its execution seamlessly 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, i.e., the programmer has neither control nor visibility of migration [3, 23, 76]. Other systems provide some form of control over migration. For example, in the Locus system [93] process migration can be triggered either by an external signal or by the 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 systems providing migration at the process level. For example, the Emerald system [47] 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 object locations and may request explicitly the migration of an object to a particular node. An example of system providing transparent migration is the COOL [57] object-oriented subsystem. COOL is built on top of the Chorus operating system [82] and is able to move objects among address spaces without user intervention or knowledge.

2.2 The mobile code approach

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. These results have been taken as a starting point by a new breed of systems providing different forms of code mobility. These systems, often referred to as *Mobile Code Systems* (MCSs), exhibit several innovations with respect to the existing approaches:

Code mobility is exploited on an Internet-scale. Distributed systems providing process or object migration have been designed with small-scale computer networks in mind, thus assuming high bandwidth, small predictable latency, trust, and often homogeneity. Mobile code systems address instead a large scale setting where networks are composed of heterogeneous hosts, managed by different authorities with different levels of trust, and connected by links with different bandwidths ranging from wireless slow connections to 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 remote nodes. The underlying run-time support provides basic functionalities (e.g., data marshaling, code check-in, and security), but it 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 exploit mobility to achieve various properties such as service customization, dynamic extension of application functionality, autonomy, fault tolerance, support for disconnected operations.

2.3 A mobile code framework

Many MCSs have been proposed. This lively and sometimes chaotic scenario has generated some confusion about concepts, abstractions, terms, and semantics of code mobility.

First of all, the fundamental distinction among implementation technologies, specific applications, and paradigms used to design these applications is not always made clear. In an early and yet valuable assessment of code mobility [39] the authors analyze and compare issues that belong to different abstraction levels. Similarly, in a recent work about autonomous objects [10], *mechanisms* like code remote evaluation (REV) [87] and RPC [11] are compared to the Echo distributed *algorithms* [19], to *applications* like "intelligent e-mail" and Web browsers, and to *paradigms* for structuring distributed applications, like mobile agents. These are different concepts that belong to different levels of abstraction and therefore cannot be compared directly. It is almost like comparing the `emacs` editor with the `fork` UNIX system call and the client-server design paradigm.

There is also confusion on terms. For instance, several systems [46, 72] claim to be able to move the *state* of a component together with its code, based on the fact that they 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. This is quite different from the situation where the run-time image of the component is transferred as a whole, including its execution state, i.e. program counter, call stack, etc. In the former case, it is the programmer's task to rebuild the execution state of 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 out from the excessive overload of the term "mobile agent". This term is used with different and somewhat overlapping meanings both in the distributed systems and in the 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 [106] an agent is represented by a thread that can migrate among different nodes carrying its execution state, in TACOMA [46] agents are just code fragments associated with initialization data that can be shipped to a remote host, without the ability to migrate again while in execution. On the other hand, in the artificial intelligence community the term "agent" denotes a software[1] component that is able to achieve a goal by performing actions and reacting to events in a dynamic environment [59]. This component is directed by knowledge of the relationships among events, actions, and goals. Moreover, knowledge can be exchanged with other agents, or increased by some inferential activity [31]. Although mobility is not the most characterizing aspect of these entities [109] there is a tendency to blend this notion of intelligent agent with the one coming from distributed systems and thus assume implicitly that a mobile agent is also intelligent and vice versa. This is actually creating confusion about between the layer providing mobility and the one exploiting mobility.

Finally, there is no definition or agreement about the distinguishing characteristics of languages supporting code mobility. In [50], Knabe lists the essential characteristics of a mobile code language, which 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 [73, 39] 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. As another example, there is no agreement on the suitable type system for mobile code languages. For example, while Knabe suggests that strong typing is a desirable property for mobile code languages, Ousterhout [70] suggests that untyped scripting languages are the best choice for this kind of systems.

Certainly, confusion and disagreement are typical of a new and still imma-

[1]For the purpose of what discussed in this paper we ignore the implications of broader notions of agent which are not restricted to the software domain.

ture 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 no 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 and 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 thesis. 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 following chapter. Mobile code technologies are used by the application developer in the implementation stage. Design paradigms are the *architectural styles* that the application designer uses as building blocks in defining the application architecture. An architectural style identifies a specific configuration for the components 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.

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, proving useful in accommodating the current situation of the field. Chapter 3 presents a general model and a classification of the mechanisms provided by mobile code technologies; mobility, communication, execution, and security mechanisms are analyzed. The classification is then used to survey and characterize several MCSs. Chapter 4 introduces mobile code design paradigms and discusses their relationships with mobile code technologies. Chapter 5 discusses the advantages of the mobile code approach and presents some application domains that are supposed to benefit from the use of some form of code mobility. In Chapter 6 we exemplify the use of our classification as a guideline for the development process by applying it to a case study in the information retrieval application domain.

Chapter 3

Mobile Code Technologies

3.1 Introduction

The interest in code mobility has been raised by the availability of a new breed of technologies featuring the ability to move portions of application code and possibly the corresponding state among the nodes of a wide-area network. These technologies apparently provide very diverse abstractions and primitives. In this chapter we analyze, classify, and compare different mechanisms provided by existing technologies, which are constituted mainly by programming languages and the corresponding run-time support.

To provide a common framework for the discussion, in Section 3.2 we discuss the differences between traditional distributed systems and MCSs by comparing the underlying virtual machines. Then, in Sections 3.3-3.6 we discuss the mechanisms exploited by existing systems to support mobility, execution, communication, and security—the key issues in MCSs. Finally, in Section 3.7 the proposed classification is used to characterize the features provided by several existing MCSs. The classification accommodates several technologies found in literature. The set of technologies considered is not exhaustive, and is constrained by space and by the focus of the thesis. However, the reader may actually verify the soundness of the taxonomy by applying it to other MCSs not considered here, like the ones described in [108, 30, 49]. The reader interested in a more detailed analysis of the linguistic problems posed by the introduction of mobility in programming languages is directed to [102, 21].

3.2 A distributed virtual machine

Traditional distributed systems can be accommodated in the virtual machine shown on the left-hand side of Figure 3.1. 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 the host targeted by communication explicitly. For instance, socket services can be regarded as belonging to

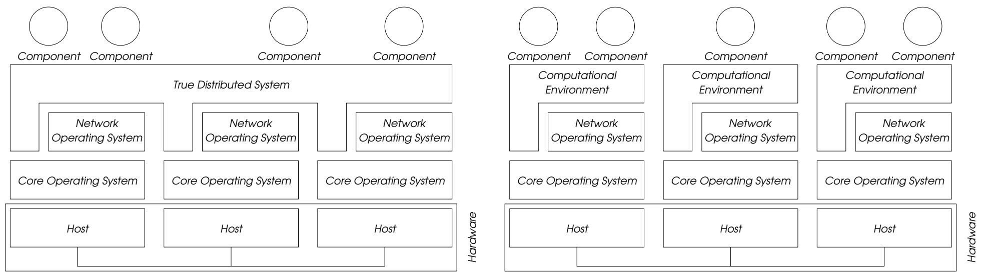

Figure 3.1: Traditional systems vs. MCSs. Traditional systems, on the left hand side, may provide a TDS layer that hides the distribution from the programmer. Technologies supporting code mobility, on the right hand side, explicitly represent the location concept, thus the programmer needs to specify *where*—i.e. in which CE—a computation has to take place.

the NOS layer, since a socket must be opened by specifying a destination network node explicitly. The NOS, at least conceptually, uses the services provided by the COS, e.g. memory management. Research on distributed systems has traditionally focused on achieving network transparency via a *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 [66] services can be regarded as TDS services since a CORBA programmer is usually unaware of the network topology and 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 3.1 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.

To give uniformity to our treatment, 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 which 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 3.2 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 components that

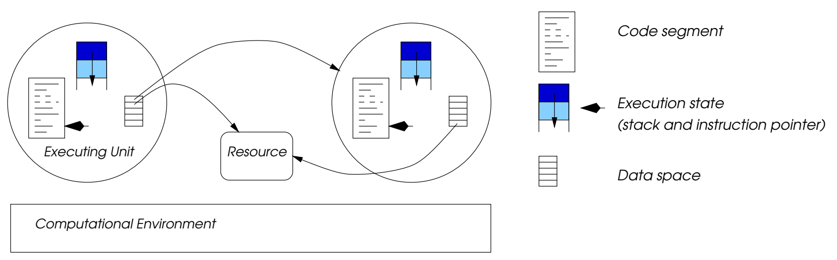

Figure 3.2: The internal structure of an executing unit.

can be accessed by the EU. As it will be explained later, these components 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.

Given the above reference model we now use the abstractions introduced so far to analyze four key areas of mobile code technology. First, and most important, we classify the mechanisms used to move either parts of or whole EUs between CEs. Then we focus on the mechanisms used to support EUs execution. Third, we analyze some of the communication mechanisms used by existing MCSs. Finally we classify the security mechanisms that have been proposed for protecting EUs and CEs from hostile entities.

3.3 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 belongs to the local CE. This is not true for MCSs. In MCSs, the code segment, the execution state, and the data space of an EU can be relocated to a different CE. In principle, each of these EU constituents might move independently. However, we will limit our discussion to the alternatives adopted by existing systems. Figure 3.3 presents a classification of mobility mechanisms.

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.

3.3.1 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*)

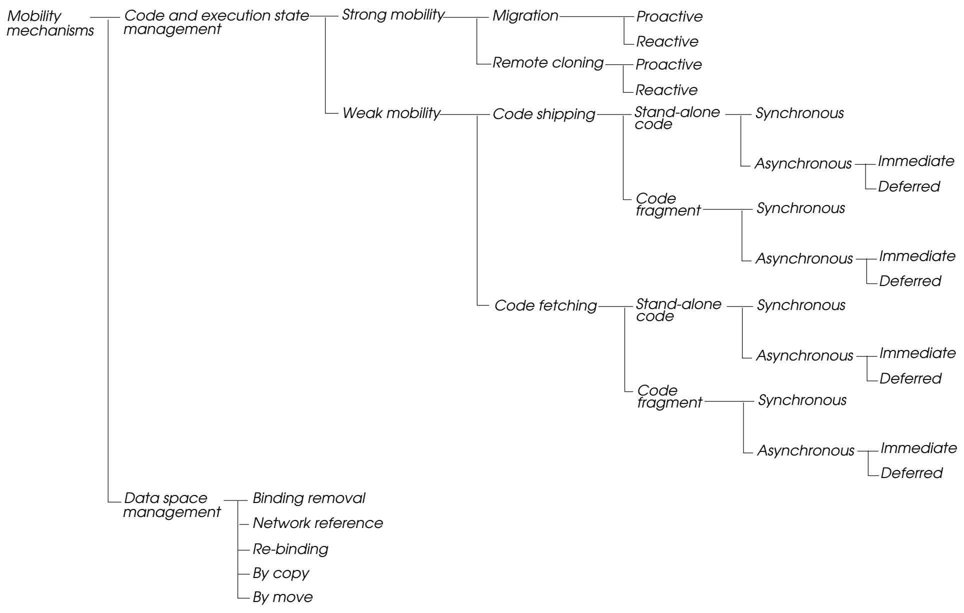

Figure 3.3: A classification of mobility mechanisms.

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. It 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 actual execution of the code transferred may take place either in an *immediate* or *deferred* fashion. In the first case, the code is executed as soon as it is received, while in a deferred scheme execution is performed only when a given condition is satisfied—e.g., upon first invocation of a portion of the code fragment or as a consequence of an application event.

3.3.2 Data Space Management

Upon migration of an EU to a new CE, its data space, i.e. the set of bindings to components accessible by the EU, must be rearranged. This may involve voiding bindings to resources, reestablishing 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 $\mathit{Resource} = \langle I, V, T \rangle$, 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 equivalently by some other 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 resource migrated 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

	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 3.1: Bindings, resources and data space management mechanisms.

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 discussion above evidences that two classes of problems must be addressed by data space management mechanisms upon migration of an EU: resource relocation and binding reconfiguration. The way 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 3.1.

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 3.4a). 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 3.4b). 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 3.4a). 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 3.4a).

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

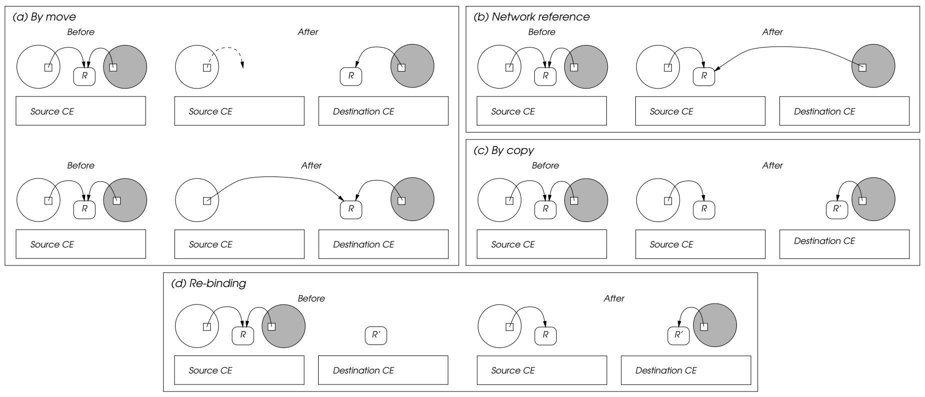

Figure 3.4: Data space management mechanisms. For each mechanism, the configuration of bindings before and after migration of the grayed EU is shown.

relevant. In this case, a copy R' of R is created, the binding to R is modified in order to refer to R', and then R' is transferred to the destination CE along with U (see Figure 3.4c). 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 possible 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 3.4d). 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. Section 3.7 will provide more insights about mobility mechanisms in existing MCSs.

3.4 Execution and translation mechanisms

The choice of executing a program either through direct interpretation or through compilation is not distinctive of MCSs. Nonetheless, the nature of MCSs introduces new requirements as well as new degrees of freedom, which affect the

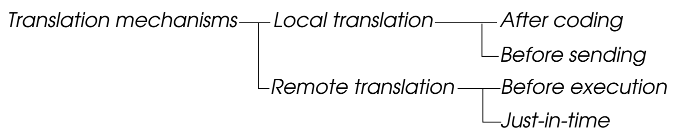

Figure 3.5: A classification of translation mechanisms.

criteria determining the choice. Code migrated among MCSs should be:

Portable The target platform is a computer network made of heterogeneous architectures. The obvious goal is to write the mobile code once, and then be able to unleash it for execution on any machine, without being aware of the specific hardware and software requirements.

Secure Code being executed within a CE must be checked, in order to protect the system from malicious or accidental damage, as discussed in Section 3.6.

As for security, interpretation enables load-time or run-time checks on the source code in order to verify that only legal instructions are executed. Interpretation is exploited frequently in MCSs because it supports portability as well, given that an interpreter is provided for each participating platform. A compiled approach, instead, would force the run-time support of the sender machine to be aware of the platform of the receiver machine in order to select the appropriate native executable code for transmission—or, if the destination platform is unknown, to send the native code for each platform. A common strategy among MCSs is the adoption of a hybrid approach in which source programs are compiled into an *intermediate language* that is then interpreted on the target machine. Using this approach, higher-level source code is compiled into a lower-level intermediate, portable code, designed to improve efficiency and safety of transmission and execution.

Using this technique, some MCSs push even further the portability issue, by achieving language independence in addition to platform independence. In this setting, CEs may support the execution of EUs whose code is written using different languages. Each CE manages several *virtual machines* each specialized for the execution of programs written in a particular language. A MCSs featuring more than one virtual machine, called a *multi-language MCS*, is a generalization of conventional MCSs where the language is fixed, e.g., Java or Telescript. Hence, we will focus our discussion on multi-language MCSs.

The basic problem of multi-language MCSs is the following: given a program written in a language L, how can it be executed on a platform that supports a set of languages $\{L_1, L_2, \ldots, L_n\}$? The solution is given by two composable mechanisms: *local translation* and *remote translation*.

Local translation is performed at the source CE and produces a program written in language $L' = T(L)$ that will be shipped to the remote CE. If the set of languages available at the remote site is not known in advance, the program written in L can be translated into different languages, in order to enhance the probability to find a suitable virtual machine at the remote CE. Local translation mechanisms can be further distinguished with respect to the *time* when translation is performed. A first solution, which is currently the most commonly employed, is to perform the translation once for all *after program coding*.

Clearly, this simplifies the design of the run-time support system. An alternative is to defer the translation to the moment when the code needs to be actually sent, i.e. *before transmission.* This way, the run-time support may exploit information about the set of languages that are supported by the destination CE, and perform only the appropriate translation.

Remote translation is performed at the destination CE and produces a program written in a language $T(L') = L'' \in \{L_1, L_2, \ldots, L_n\}$, whose execution is supported by the CE. Again, these mechanisms can be distinguished with respect to translation time. Translation can be performed on the whole program just *before execution*, or it can be performed *just-in-time* as soon as the execution flow reaches untranslated portions of the original program.

A real MCS does not necessarily need to exploit all the mechanisms described so far, which are summarized in Figure 3.5. For example, the common scheme where a single high-level language is compiled into a single low-level interpreted language needs just local translation after coding, which directly generates a program in the language supported on the destination CE. Section 3.7 will provide more insights about how MCSs support code translation and execution. Anyway, the way these mechanisms are composed and applied may vary, depending on a number of considerations and trade-offs related to code size, efficiency, and portability.

3.5 Communication mechanisms

Code migration can be regarded as a form of communication *per se*, which takes place among processes distributed at different locations—the CEs. As a matter of fact, the first approaches to code mobility [87, 25] where conceived as extensions of the remote procedure call communication mechanism. Furthermore, a mobile EU can be used to disseminate information among a set of CEs, thus realizing a form of communication. However, within the context of MCSs the issue is usually to enable communication among migrating EUs rather than among the CEs supporting their execution.

To this end, the alternative could in principle encompass many of the mechanisms found in literature. Nevertheless, EUs roaming in a distributed system pose a problem which is related intrinsically with mobility: some of the communication mechanisms need to be extended to work properly in a mobile setting. Message passing, for instance, assumes that the recipient of a message is identified with an address, which is statically associated with a location. Clearly, if the recipient is allowed to detach from its current location and move to a different one, new protocols must be devised to avoid message loss. Similarly, an EU migrating to a new CE may need to communicate with other EUs located there: mechanisms are needed to establish such communication. This problem is not specific to EUs moving across CEs. Similar problems arise in the context of mobile computing, where packet routing must be changed dynamically according to the movement of the physical host, which in addition needs to be aware of the facilities available at a given access point [74].

Development of communication mechanisms in MCSs is still in its early phase. This is mainly because until now research has been focused on mobility mechanisms. After the initial successes in this effort, researchers are now investigating more complex forms of interactions among EUs.

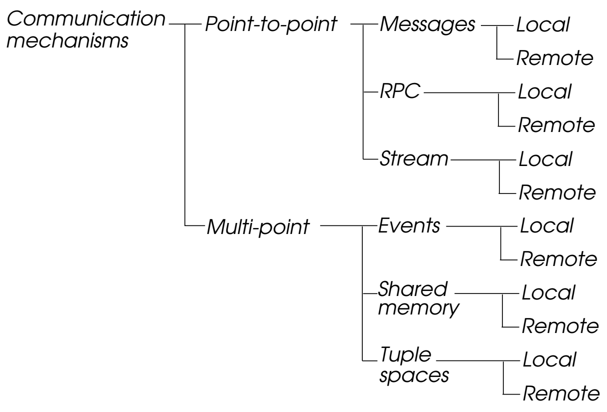

Figure 3.6: A classification of communication mechanisms.

Mechanisms supported by existing MCSs can be distinguished along two orthogonal dimensions: the number of EUs involved in communication, and their mutual location. The first dimension partitions the space of mechanisms in *point-to-point* and *multi-point* communication mechanisms. The second dimension distinguishes among *local* and *remote* mechanisms, depending on whether the EUs are co-located in the same CE or not. At a first glance, remote communication seems in contradiction with one of the goals of code mobility, i.e. to minimize communication over the network. However, as it will be discussed later, mobile code is an option rather than *the* solution, and remote communication mechanisms may be helpful even when used in conjunction with mobile EUs, e.g. to monitor EU activity or to interact with systems which do not offer support for code mobility. The classification for communication mechanisms is summarized in Figure 3.6.

Point-to-point mechanisms Point-to-point mechanisms enable communication between two EUs. The more primitive mechanism is *message passing*, although no existing MCSs actually uses it for communication between EUs moving arbitrarily, due to the problems arising in presence of mobility, as discussed earlier. Usually, some form of *remote procedure call* [11] (RPC) is provided instead. This may involve invocation of bare procedures or method invocation on remote objects, a la CORBA. RPC is often exploited for both local and remote communication, in order to provide a uniform mechanism. *Streams* are a different communication mechanism that allows an EU to open a channel to another EU, and send it a continuous stream of data. Like RPC, streams are often used for local communication as well, much like pipes in a Unix system. Notably, some MCSs manage to retain open streams as part of the state of an EU during the migration process, thus providing primitives that abstract from the physical location of the EUs involved in communication.

Multi-point mechanisms Multi-point mechanisms enable communication among more than two EUs at a time. *Shared memory* is by far the mechanism most frequently used to achieve multi-point communication. In order to communicate, several EUs are given a reference to the same variable or object.

Changes in the value associated with the object are perceived by all the EUs owning a reference to it. In existing MCSs, this mechanism is always used for local communication, although it could be extended to implement remote communication, for example with a distributed shared variable abstraction, as described in [4].

Event-based mechanisms, in turn, define an event bus constituting the logical channel through which events are dispatched, together with primitives that allow EUs to generate events and to subscribe for receiving events they are interested in. Upon generation of an event, the communication systems sends a copy of the event description to all the EUs that subscribed to it. Hence, events provides a form of *implicit* communication. Variants of this scheme exist in MCSs. Many languages exploit this mechanism for local communication only—events are dispatched only to EUs within the CE where the event is generated. In fact, event dispatching suffers of the same problem highlighted for message passing in presence of mobile EUs. Furthermore, building scalable and efficient mechanisms for dispatching events over large-scale networks is still an open research area [79, 80]. Nevertheless, the first proposals of architectures overcoming the problem are beginning to appear in [6, 108, 22]. Some of these proposals combine the notion of event with the notion of *group*. The relationship determining whether an EU belongs or not to a given group is not necessarily related with communication: for instance, it can be used to specify that a set of EUs are under the same authority. In the context of communication, groups provide a naming scheme which can be used to support multicast or anycast schemes in message or event-based communication. A typical example is the following. Assume a group of EUs have been unleashed over the Internet to retrieve information. If an EU finds an information item, it might be desirable to inform the other EUs of this fact and communicate them that the next item has to be searched instead. The group naming scheme and a distributed event mechanism may provide a good set of abstractions to accomplish this.

Another form of implicit communication is provided by *tuple spaces*. With this mechanism [17], EUs communicate by either inserting the tuples containing the information to be communicated into a shared tuple space, or by searching it for a tuple using some form of pattern matching. This mechanism provides a very powerful abstraction of a communication channel. However, its application to a large-scale setting is limited by the implementation problems that arise in maintaining a notion of shared tuple space across the network. For this reason, remote communication through tuple space is not implemented by any MCSs, yet. Instead, tuple spaces are exploited as a mechanism for local communication.

3.6 Security mechanisms

Security is a big concern in systems providing support to code mobility and is often considered to be the main limitation to the wide acceptance of mobile code technologies [99]. MCSs provide a distributed computing infrastructure on which applications belonging to different (usually untrusted) users can execute concurrently. In addition, the CEs that compose the infrastructure may be managed by different authorities (e.g. a university or a company) with different and possibly conflicting objectives and may communicate across untrusted communication infrastructures, e.g. the Internet. In this scenario several attacks are possible. Unauthorized people may eavesdrop network traffic to access or modify application data while they are in transit over the network. Applications may try to masquerade as privileged applications or claim to belong to particular unaware users when trying to move code to remote CEs. A CE, on the other side, could try to impersonate another legitimate CE in order to receive application data and gain access to application private information.

Even if strong authentication may protect from these attacks, applications may try to build several kinds of attacks against the CEs supporting their execution. First of all, an application may try to gain privileged or unrestricted access to private information belonging to the CE (e.g., its secret key) or to the underlying operating system (e.g., the password file of a UNIX machine). In addition, an application may try to misuse services offered by the CE to attack the CE itself or to use the CE as the basis for probing and exploiting the vulnerabilities of other systems. For example, unrestricted access to the socket facility of the operating system underlying the CE could be used to attack hosts behind a firewall. As another example, unregulated access to the file system may allow an application to install virii or trojan horses. Unrestricted access to CE resources may also allow an application to build denial-of-service attacks, in which an application tries to jeopardize the functionalities of a CE by exhausting its memory, storage system, or CPU cycles. Similarly, applications may try to disclose information, misuse services, or deny normal execution of other applications. For example, if different applications share the same address space, an application could examine the data or code of other applications to steal strategic information or tamper with their execution. An application may even try to induce another application to execute arbitrary code that carries out malicious operations, or to prevent the application from completing its execution by flooding it with messages or simply by requesting its termination.

These problems have been studied by the distributed systems and computer security communities for a long time. Many of the results achieved in those fields can be used in order to secure mobile code systems. Unfortunately, MCSs introduce a completely new security issue: since application or their EUs may travel across different CEs with different levels of trust, CEs could try to tamper with EUs code or state in order to disclose private information associated with applications, gain competitive advantage with respect to other CEs, or modify applications code so that they will attack other CEs or applications. Consider for example an application composed of an EU that roams the network looking for the best offer for a compact disc. A CE executing the EU may modify the EU data structures containing the prices offered by other companies and then change the code of the agent so that information collected by visiting other CEs will be ignored during the determination of the best offer.

The scenarios described so far suggests that security issues could be divided into two classes: *intra-CE security*, i.e., the issues related to the problem of protecting entities inside a CE, and *inter-CE security* that includes the issues raised by interactions among remote entities. The classification of security mechanisms is shown in Figure 3.7.

Inter-CE security Inter-CE security issues encompass authentication, integrity, and privacy of the communication between two CEs, an EU and a remote CE, and two remote EUs.

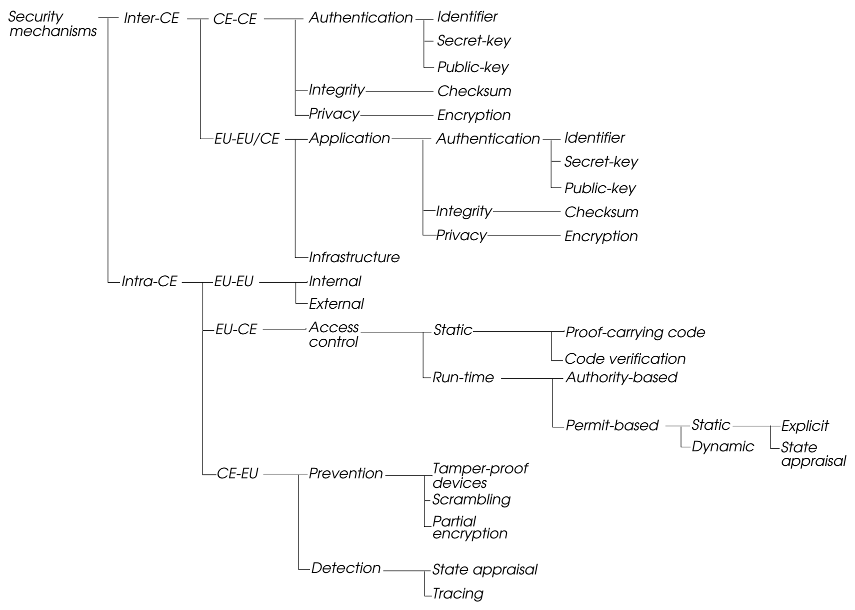

Figure 3.7: A classification of security mechanisms.

When interacting, CEs need to authenticate each other in order to be protected from spoofing [20]. Authentication mechanisms can be based simply on identifiers (network addresses or names) that provide little or no security or may use some stronger mechanism based on cryptography [84], namely, secret-key authentication systems (a la Kerberos [88]), or authentication systems based on public-key cryptography and digital signatures, like the ones conforming to the X.509 standard [45]. In addition, authentication needs to be accompanied by mechanisms that provide for integrity and privacy of the exchanged data, in particular of code or EUs that are moved from CE to CE. Integrity mechanisms ensure that transmitted data is not modified, either by malicious intent or by errors in the transmission process, using cryptographic checksums (e.g. MD5 [77]). Privacy mechanisms exploit encryption (e.g. DES [67]) to ensure that third parties not involved in inter-CE communication cannot access the information transmitted over the network.

Similar to the previous case, interactions between an EU and a remote CE or between two remote EUs requires mechanisms to authenticate the partners of communication and to protect data during transmission over an untrusted channel. These mechanisms can be implemented at the application level including security information in the data being exchanged (e.g. agent id and IP address or cryptographic checksums and digital signatures) or can be mediated by the underlying CEs that use CE-to-CE security mechanisms in order to protect EU-to-CE or EU-to-EU communication.

Intra-CE security Intra-CE security addresses the problem of protecting the entities that compose a single computational environment. Intra-CE security encompasses security among different EUs, protection of the hosting CE and the associated resources from EUs, and protection of EUs from the CE. Most of these issues are addressed by *authorization* that determines the acceptable actions of an entity on the basis of its identity, as determined by the authentication process.

EUs hosted by a CE must be protected by unauthorized or malicious attempts to tamper with each other. Protection is achieved by means of access control mechanisms. These mechanisms can be *internal* or *external*. Internal mechanisms allow an EU to examine requests for services and explicitly grant access on the basis of the identity of the requesting EU and the service parameters. For example, an internal communication security mechanism may act as a wrapper, allowing an EU to examine the identity of the partner that requested the setup of a communication channel before actually accepting the connection. External security mechanisms limit the actions of an EU with respect to another EU by using some information maintained in the CE that specifies the associated security policy for EU interactions. For example an EU may configure the CE access control mechanisms so that only EUs of a certain group are allowed to stop its execution. Requests of termination coming from EUs outside this group would be intercepted and blocked by the CE before they reach the intended EU.

As in the previous case, the resources and functionalities of a CE are protected from attacks or faulty actions of incoming EUs by means of access control mechanisms that, given an EU, determine *which* resources can be accessed and *how much* of them the EU can use. The aim is to protect CEs from denial-of-service attacks that, by exhausting their resources, prevent them from functioning properly. Access control can be performed *statically* or *at run-time*. Static access control mechanisms aim at determining, before EU execution, if, given the incoming EU code and current state, execution will respect the access control policy enforced by the system. If the verification succeeds, no other check will be performed during EU execution. For example the *proof-carrying code* [64] mechanism associates code fragments with a formal proof of their correctness that is verified before code execution. A different approach is followed by *code verifiers* that use theorem proving techniques to determine if the UE's code preserves some security invariants (e.g., address space containment).

In run-time access control mechanisms, each EU has an associated set of access rights to local resources (e.g. permission to read/write a file or to use a certain amount of CPU time) that is determined by the security policy of the CE. Every attempt of an EU to access CE resources is intercepted and examined with respect to the access rights of the EU. As a consequence of the check, the required operation can be performed or denied. This access control mechanism is often called *safe execution* or *sandboxing*. Run-time access control mechanisms can be distinguished on the basis of the information used to determine the set of access rights of an EU. *Authority-based* mechanisms determine access rights on the basis of the authority under which the EU is running. *Permit-based* mechanisms determine access rights on the basis of a *permit* that is associated to the EU. Permits can be statically associated to the EU for its whole lifetime or they can be determined dynamically. For example the *state appraisal* mechanism [26] defines an EU permit on the basis of a state appraisal function that is associated with the migrating EU. When the EU reaches a new execution environment the appraisal function is evaluated passing as parameter the EU current state, as it has been defined by execution at other CEs. The function returns the set of access rights that the EU needs in its current state. On the basis of the requested access rights, the receiving

CE may allow or deny the execution of the EU. Other dynamic permit-based mechanisms allow an EU to re-contract its access rights during execution using some kind of electronic currency.

One of the most difficult problem in intra-CE security is represented by protection of an EU from the hosting CE. Attacks may include denial-of-service, service overcharging, private information disclosure, and code/data modification. To execute EUs, CEs must access their code and execution state. As a consequence, it is very difficult to protect EUs from malicious CEs. Protection mechanisms can be aimed at *prevention* or *detection*. Prevention mechanisms try to make it impossible to access or modify an EU data and code in a useful way. For example, *tamper-proof devices* [107] avoid unauthorized modification of EU code and state by executing EUs in a physically sealed environment that is not accessible even by the owner of the system without disrupting the system itself. A different approach is followed by the code scrambling mechanism, in which the EU code is rearranged before EU moves to a CE in order to make almost impossible reverse-engineering [41, 42] while maintaining the EU's original behavior. A simpler solution performs *partial encryption* of the EU contents. Using this mechanism, an EU could protect data that has to be used in a particular CE by encrypting it with the CE's public-key. This way data will be accessible only when the EU reaches the intended execution environment. Other mechanisms that exploit execution of cryptographed functions are being studied [83]. Detection mechanisms aim at detecting, after EU execution, if some illegal modification of its code and state occurred in the EU lifetime. A first mechanism is based on the state appraisal functions described earlier. State appraisal is used after execution to detect illegal modifications in the EU state that invalidate some invariants associated with the EU. A second mechanism is *cryptographic tracing* [97, 98] that, using execution traces and digital signatures, allows the owner of an EU to detect, after EU termination, every possible illegal modification of EU's code and state. Cryptographic tracing will be described in detail in Chapter 7.

As a matter of fact, usually MCSs consider CEs as trusted entities and the problem of protecting EUs form CEs is not tackled.

3.7 A survey of mobile code systems

Currently available technologies differ in the mechanisms they provide to support mobility, execution, communication, and security. In this section we use the mechanism classification presented so far to characterize a number of mobile code technologies. Each technology is first mapped onto the virtual machine introduced in Section 3.2 and then its characteristics are analyzed with respect to mobility, translation and execution, communication, and security.

Agent Tcl (`http://www.cs.dartmouth.edu/~agent/`)

Developed at the University of Darthmouth, Agent Tcl [36] 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 plus the language run-time support. The basic functional unit of run-time support is the *agent server*, that implements inter-agent communication together with agent check-in and check-out services. Some extensions are planned for the language, in order to support disconnected users and to improve security. Our analysis is based on version 1.1 of the language that does not feature these additional functionalities.

Mobility 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. Such mechanism is asynchronous and immediate. A copy of variables belonging to the execution state of the EU invoking the `submit` may be migrated together with the Tcl script by passing them explicitly as parameters.

Translation The design of the Agent Tcl language is aimed at supporting different languages. Presently Tcl is the only language supported. As a consequence, no translation mechanisms are needed. Agents are transferred from site to site in source code form. A new version of the language, called D'Agents [35] is under development. This implementation will support agents written in Java, Tcl, and Scheme.

Communication Two communication mechanisms are provided: *messages* and *meetings*. Messages allow EUs to exchange messages composed of strings. The message recipient can be located in the same CE or in a remote CE. Meetings model, at the language level, socket-like stream connections between agents. An EU can request a meeting to another agent either in the same CE or in a remote CE. In the current implementation, meetings are not preserved upon migration.

Security In Agent Tcl a CE can be configured in order to accept mobile EUs from selected hosts only. Apart from this simple identifier-based CE-CE security mechanism, Agent Tcl provides no other inter-CE security mechanisms. An extension that allows EUs and CEs to authenticate each other and to support privacy and integrity using PGP [112] is under development. Inside the CE there is no mechanism to protect EUs from each other and to protect EUs from CEs. Protection of the CE from EUs is delegated to the underlying operating system mechanisms, and therefore EUs may access any resource that is accessible to the principal that is responsible for the execution of the run-time support.

Aglets (`http://www.trl.ibm.co.jp/aglets`)

The Java Aglets API (J-AAPI) [55], developed by IBM Tokyo Research Laboratory in Japan, extends Java with support for weak mobility. Aglets [56], 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.

Mobility 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.

Translation Aglets are Java programs that are compiled into Java Bytecode after development. Therefore the Aglet system exploits a "local translation after coding" mechanism.

Communication Aglets can communicate through *messages*. The Aglet system leverage off of the message concept in order to implement both a point-to-point and a multi-point communication mechanism. In the former case, messages are used to implement an RPC-like communication mechanism in which messages are exchanged following a request/reply schema. The mechanism can be local or remote. In the latter case, messages are used to implement a local event-based communication system that allows an EU to send a message to several EUs in the same context, provided that the EUs subscribed for the message.

Security In the current aglet implementation, Inter-CE security is achieved by a simple identifier-based authentication mechanism. This mechanism allows a CE manager to select the hosts that can submit aglets. Extensions for achieving integrity adn privacy in inter-CE interactions are under development.

Aglets are protected [48] from each other by using an internal security mechanism. Operations invoked on an aglet are dispatched to a *proxy object* that acts as a wrapper for the actual aglet, that can take actions based on the information associated with the operation request before executing the corresponding operation. For example, upon invocation of a `dispatch` operation on an aglet proxy, a predefined sequence of method invocations (whose code is under the control of the programmer) is automatically started, which allows the aglet to determine whether the request has to be accepted, rejected, or delayed. This mechanism protects both local and remote EU-EU interactions.

The CE is protected from EUs using an access control mechanism. The CE can be configured to grant particular capabilities (access to the file system, access to the windowing system, and access to network) to aglets on the basis of their source. The mechanism is therefore static and authority-based.

Ara (`http://www.uni-kl.de/AG-Nehmer/Ara/`)

Developed at University of Kaiserslautern, Ara [73] 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 In Ara, mobility is supported through proactive migration only. Data space management is simplified by the fact that agents cannot share anything but system resources—whose bindings are always removed upon migration.

Translation Ara agents may be written in different languages. Tcl agents are not translated and, as in the case of Agent Tcl, they are transferred in source code form. C and C++ agents are translated, at the starting site, in a common lower-level language called MACE, after coding.

Communication Communication in Ara is only local and is achieved by means of simple messaging. Server agents can register a service point with a well-known name in a place and client agents can send messages to the service point to have them delivered to the server. A mechanism supporting messaging among EUs running on different CEs is under development.

Security Ara does not provide any cryptographically strong mean to authenticate and protect agents moving from site to site. CEs are protected from EUs using a static, explicit, permit-based, run-time, access-control mechanism, called *allowance*. An allowance specifies a set of access rights to the system resources, like files, network connections, and CPU time. An agent migrating to a site carries an allowance specifying the desired access to the target place. In turn, the place, before starting the agent, determines an actual allowance for the agent based on the requested allowance and the place security policy. Since EUs can interact by message passing only, EU-EU security is achieved using internal mechanisms, i.e. authentication, integrity, and privacy are managed at the application level by the interacting EUs.

Facile (`http://www.ecrc.de/research/projects/facile/`)

Developed at ECRC in Münich, Facile [94] is a functional language that extends the Standard ML language with primitives for distribution, concurrency and communication. The language has been extended further in [50] to support weak mobility. Executing units are implemented as threads that run in Facile CEs, called *nodes*.

Mobility and communication In Facile, mobility is achieved exploiting the *channel* communication mechanism provided by the language. Channels can be used to communicate, locally or remotely, any legal value of the Facile language between two threads. 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.

Translation The CE is able to handle several kinds of intermediate formats for the executable, spanning from source code to native binaries. Therefore, the language implements a translation after coding mechanism. In addition Facile provides a remote just-in-time mechanism that support dynamic compilation of source code into platform-native format.

Security No dedicated security mechanisms are provided at the time of writing.

Java (`http://java.sun.com/`)

Developed by Sun Microsystems, Java [62] 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.

Mobility 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, independently of the need to execute the class code. Therefore Java supports weak mobility using mechanisms for fetching of 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 the World Wide Web technology. Web browsers have been extended to include a JVM, and particular Java classes called *applets* can be downloaded together with HTML pages to allow for active presentation of information and interactive access to a server. We consider this as a particular *application* of mobile code technology. On the other hand, the combination of a Web browser and a JVM is so frequent that it can be regarded as a technology per se, supporting the development of Web applications. The presence of a JVM is hidden and its mechanisms are used to provide a higher-level view 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 even if, in principle, it is based on the code fragment code fetching mechanism provided by the underlying JVM.

Communication Being conceived for the implementation of networked applications, Java is a language with a rich set of communication-oriented APIs. Support for messaging, and stream communication mechanism is provided. In addition, Java features an RPC-like mechanism called Remote Method Invocation (RMI) [90], that allows a Java application to invoke methods exported by other applications. All the mechanisms are both local and remote. Mechanisms for communication based on events and tuple spaces are under development.

Security As for security, the Java Byte Code received from the network is not directly interpreted by the Java Virtual Machine. Instead, it is first scanned by a *Byte Code Verifier*, to check for the absence of potentially dangerous constructs and then, during execution, it is controlled by a programmable *Security Manager*. The former mechanism is a static access control mechanism based on code verification, while the latter is a run-time programmable sandboxing mechanism. Since this mechanism is programmable, both authority-based and permit-based policies can be easily implemented.

M0 `(http://cuiwww.unige.ch/Telecom-group/msgr/home.html)`

Implemented at the University of Geneva, M0 [95] 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.

Mobility Messengers [96], 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, in order 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 by copy.

Translation M0 is a postscript-like interpreted language and messengers are shipped in source code form. Therefore no translation mechanism is needed.

Communication Communication among messenger is achieved using a local shared associative array (called *dictionary*), similar to a tuple space.

Security Integrity of messengers transmitted over channels is supported via checksums included in the messenger message, but no mechanisms for CE-CE authentication or privacy is provided. Inside a CE, M0 provides effective privacy and run-time, authority-based access control mechanisms. References to shared objects may have *attributes* that specify Unix-like access rights for a messenger with respect to the referenced object. In addition, messengers may use asymmetric encryption to create protected entries in shared memory, accessible by means of a public key, but modifiable only by means of a private key.

Mole (`http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole.html`)

Developed at University of Stuttgart, Mole [89, 6] 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.

Mobility Mobility is supported by a mechanism for the asynchronous shipping of stand-alone code, which is executed immediately. The code and data to be sent are determined automatically upon migration using the notion of island [40]. 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.

Translation Mole agents are Java programs. Therefore, as for the Aglet system, a "local translation after coding" mechanism is adopted.

Communication The Mole project puts a strong emphasis on inter-agent communication which, according to the description in [6] is centered around three main concepts: agent groups, sessions, and events. Agent groups can be created by defining a *badge*, i.e. an application-dependent identifier which an agent can "pin" on and off. Predicates on badge values can be specified within the application code, thus enabling the dynamic creation of groups of related agents and (non-deterministic) selection of their elements for communication. Actual communication, supported either through message passing or RPC, requires the setup of a session. During this phase, each agent can specify its will to participate in the session with either blocking or non-blocking operations; moreover, it describes the agent which it intends to communicate with by means of a pair `(placeId, PeerQualifier)`, where the second component can be either a globally assigned agent identifier or a badge predicate. Finally, anonymous communication is encompassed by event management, which is still under design.

Security As far as security is concerned, Mole does not provide anything but the underlying Java mechanisms.

Obliq (`http://www.luca.demon.co.uk/Obliq/Obliq.html`)

Developed at DEC, Obliq [16] is an untyped, object-based, lexically scoped, interpreted language. Obliq allows threads, the Obliq EUs, to request the remote execution of procedures to *execution engines* which implement the CE concept.

Mobility When a thread requests 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. Access to these objects is translated into callbacks to the originating CE. This sharing strategy makes transparent the actual location of the elements referenced in the EU data space, but the use of network references may result in complex debugging and performance bottlenecks.

Translation Obliq is an interpreted language and code is shipped between CEs in source code form, and no translation occurs.

Communication Obliq threads may communicate, both locally and remotely, by method invocation or using shared memory.

Security In Obliq there are no explicit security mechanisms. Lexical scope rules support an implicit form of safe execution. In fact, procedures evaluated by an Obliq engine, by default, may only access objects in their original scope. Therefore, access to objects and files residing at the engine CE must be granted explicitly by the remote engine passing a reference to a local component when invoking the procedure.

Safe-Tcl (`http://sunscript.sun.com`)

Initially developed by the authors of the Internet MIME standard, Safe-Tcl [13] is an extension of Tcl [69] conceived to support active e-mail.

Mobility 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.

Translation Safe-Tcl scripts are transferred in source code form using the MIME standard format and no translation occurs.

Security In Safe-Tcl the focus of the language is on the design of access control mechanisms to protect the recipient of a Safe-Tcl script. This 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 [71], much like Java applets

Sumatra (http://www.cs.umd.edu/users/saltz/)

Sumatra [2], 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. They are executed within *execution engines*, i.e. extended Java interpreters which implement the CE abstraction.

Mobility Sumatra provides mechanisms for proactive migration, 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 which 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.

Translation Sumatra is based on Java and therefore exploits a "local translation after coding" mechanism that translates Java programs into Java ByteCode.

Communication Remote access is achieved through the Java RMI API, that implements an RPC-like point-to-point mechanism. An additional communication mechanism is represented by *asynchronous events*, which are provided mainly to support *resource monitoring*, the main focus in Sumatra. Differently from other approaches, events are dispatched only locally, in order to inform a thread of resource changes or other site-related information, and are not conceived to be used as an inter-CE communication mechanism.

Security No security features are provided.

TACOMA (http://www.cs.uit.no/DOS/Tacoma/)

In TACOMA [46] (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.

Mobility In TACOMA, 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.

Translation TACOMA is a multi-language system supporting, in version 1.2, a number of interpreted languages (namely Tcl, Python, Scheme, and Perl), and C. Code of agents written in interpreted language is shipped in source code form and no translation occurs. C source code is shipped to the remote site where is compiled and then executed. Therefore in this case a remote translation before execution mechanism is used.

Communication TACOMA agents can communicate using briefcases, but only at startup time. In fact, the `meet` primitive (whose name is at least counter-intuitive) invokes a *new* program specified by a primitive parameter, passing on the standard input the contents of a briefcase specified as another parameter. Communication between active agents can only be performed using *cabinets*, i.e. shared data containers maintained in the file system and can involve only agents residing on the same CE.

Security TACOMA provides two basic security mechanisms. First, the CE owner can specify the set of TACOMA hosts from which agents are accepted. Second, the execution of the whole system can be confined in a subtree of the host file system. Apart from these mechanisms no inter-CE or intra CE mechanisms are currently implemented.

Telescript (`http://www.genmagic.com/Telescript`)

Developed by General Magic, Telescript [106] 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's EUs are *agents* and *places*, whose execution is supported by *engines*, the Telescript CEs.

Mobility 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.

Translation Telescript employs an intermediate, portable language called *Low Telescript*, which is the representation actually transmitted among *engines*. Therefore Telescript exploits a translation after coding mechanism.

Communication Agents can interact locally after they *meet*. As a result of the meeting, each agent obtains a reference to the other. Then, both method invocation (RPC-like) and shared memory mechanisms can be used for communication.

Security Of all reviewed MCLs, Telescript provides the most powerful mechanisms to support security [91]. In Telescript, each thread object has attributes that can be used to determine its security-related characteristics. For example, attributes are provided to hold the *authority* of the thread, i.e. the real-world person or organization which it belongs to, and can be accounted for. A particular set of these attributes, called *permits*, grant the agent the right of performing a given set of operations (e.g., the `go` operation), or to use engine resources (e.g., the CPU time) in a specified amount. Permit violation is controlled by the programmer through the exception handling mechanism provided by Telescript. The values of the security attributes are partly specified by the programmer and partly set by the Telescript engine where the trip originates, or by the engine where the trip ends. The mechanism by which the engine sets these values is fully programmable by the engine owner through the methods of the engine place, that is, the particular thread that represents the engine. Agent integrity is provided by packing and encrypting the agents before their trip.

Chapter 4

Mobile Code Paradigms

4.1 Introduction

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. In this chapter we analyze how mobility affects architectural design and we introduce new design paradigms that allow the developer to take into account mobility during the definition of the software architecture of a distributed application.

4.2 Software architectures and architectural styles

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 [85]. Software architectures with similar characteristics can be represented by *architectural styles* [1] 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. A *modular* system can be built even 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.

4.3 Mobile code paradigms

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 [103], 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[1], 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 [32].

4.3.1 Basic concepts

Before introducing design paradigms we present some basic concepts that abstract out 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 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.

[1]The reader interested in the original formulation of the paradigms described here is directed to [18]. A case study centered around a formalization of these paradigms using the UNITY notation is also provided in [75].

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

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 4.1).

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.

4.3.2 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,

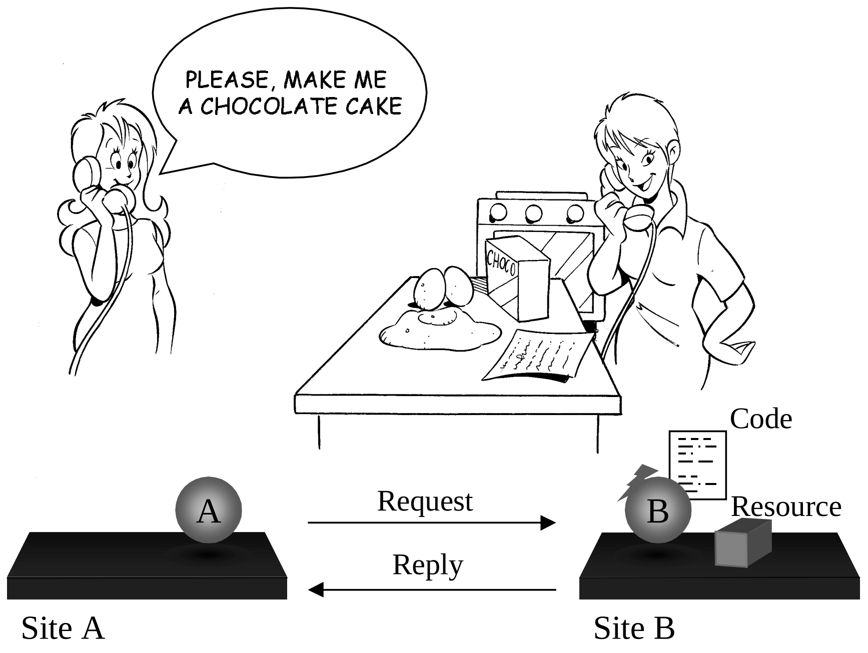

Figure 4.1: Client-server cake.

B performs the service requested by executing the corresponding know-how and accessing the involved resources co-located with *B*. In general, the service produces some sort of result that will be delivered back to the client with an additional interaction.

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

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

4.3.3 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.

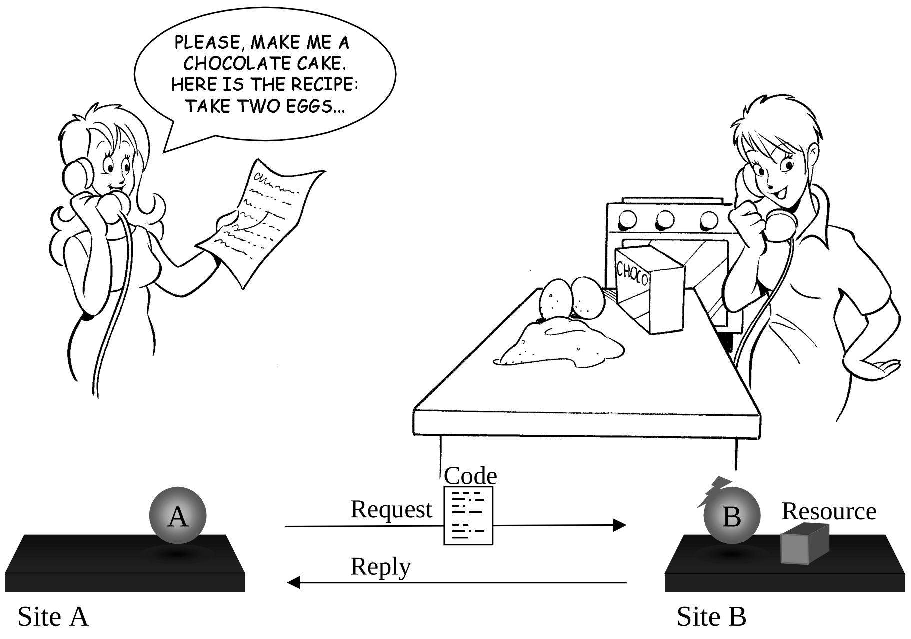
PLEASE, MAKE ME A CHOCOLATE CAKE. HERE IS THE RECIPE: TAKE TWO EGGS...
CHOCO
Code
A
Request
Reply
B
Resource
Site A
Site B

Figure 4.2: Remote evaluation cake.

In the REV paradigm[2], 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, executes the code using the resources available there. An additional interaction delivers the results back to A.

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

[2]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 [87], 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.

the remote evaluation paradigm offers a service that is programmable with a computationally complete language.

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

4.3.4 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.

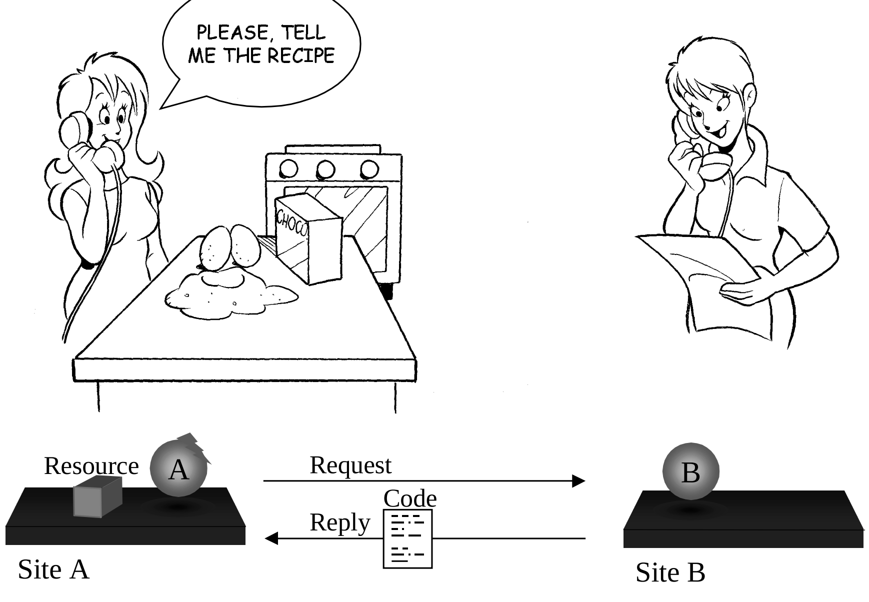

Figure 4.3: Code on demand cake.

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, which can subsequently execute it.

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

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

As an example of an application that is conveniently modeled with mobile agents, consider a workflow system in which activities are represented by applications that enclose some information (e.g., a document), the associated state (e.g., draft, revised etc.), the operations applicable in the current state (e.g., annote or modify) and the workflow actor responsible for the performance of the current task in the activity. Activities are able to move to the work environment of such user and interact with the user locally until the user's task is accomplished. Then, activities move close to the next user that has to perform the next task of the activity.

4.3.6 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 [5, 32] 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

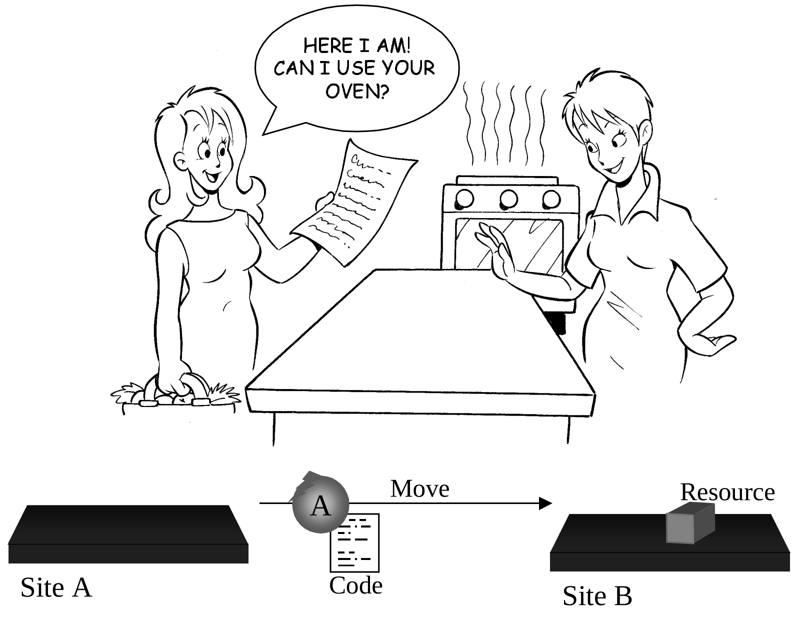

Figure 4.4: Mobile agents cake.

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, extendibility, and dynamic reconfigurability 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. We think that no paradigm 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, together with some criteria to evaluate the parameters values. For example, one may want to minimize the number of interactions, the CPU costs or the generated network traffic. In addition, a model of the underlying distributed system should be adopted to support reasoning about the criteria. For each paradigm considered, an analysis should be carried out in order to determine which paradigm optimizes the chosen criteria. This phase cannot take into account all the characteristics and constraints, which probably will be fully understood only after the detailed design, but it should provide hints about the most reasonable paradigm to follow in the design. A case study that provides guidelines on how such analysis can be carried out is given in Chapter 6.

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 interpreter 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 [8]. In this case, the architectural concept of a moving component must be implemented using a technology that allows just for mobile code modules. 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 in order 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 [32]. A case study that analyzes these relationships in detail can be found in Chapter 6.

Chapter 5

Mobile Code Applications

5.1 Introduction

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 are concerned—and of the lack of 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 what is an application (e.g., a system to control a remote telescope) and what is 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 chapter 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. Chapter 6 will show an example of how our taxonomy can be leveraged off in the application domain of distributed information retrieval.

5.2 Key benefits of mobile code

A major asset provided by code mobility is that it enables *service customization.* In conventional distributed systems built following the 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 in increasing 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 lengthy 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 from the design level. 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 [28, 43]. In networks where some regions are connected through wireless links while others are connected through conventional links the design becomes complex, in that it must cope with frequent disconnections and aim at avoiding as much as possible the generation of traffic over the low-bandwidth links. 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 and, 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 in order 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 those data.

5.3 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.

5.3.1 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 in principle could range from the list of all the publications of a given author to the software configuration of hosts in a network collected for administrative purposes. Code mobility could improve efficiency by migrating the code that performs the search process close to the (possibly huge) information base to be analyzed [51]. This type of application has been often considered "the killer application" motivating a design based on the MA paradigm. However, analysis of the generated network traffic in some typical cases evidenced that, according to the parameters of the application, the CS paradigm sometimes can be the best choice [18].

5.3.2 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 concepts are fundamental for these applications since they enable the embedding of code and state into documents, and support 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 remote databases. 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.

5.3.3 Advanced telecommunication services

Support, management, and accounting of advanced telecommunication services like videoconference, video on demand, or telemeeting, demand for a specialized "middleware" providing mechanisms for dynamic reconfiguration and user customization—benefits provided by code mobility. For example, the application components managing the setup, signaling, 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 [58] and [60]. 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 [38].

5.3.4 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 periodical polling of the resource state and configuration is performed using a predefined set of services. This approach, based on the CS paradigm, can lead to a number of problems [110]. 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 [5, 33]. A case study focused on the application of our taxonomy to the network management application domain is presented in [29].

5.3.5 Workflow management and cooperation

Workflow management applications support the cooperation of persons and tools involved in a development process. The workflow defines which activities must be carried out in order 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 [15].

5.3.6 Active networks

The idea of active networks has been proposed recently [92, 111] 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 [9], most of the approaches rely on it and can be classified along a spectrum delimited by two extremes represented by the programmable switch and the capsule approaches [92]. 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 dynamically 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 [104] is an example of an active network architecture exploiting the COD paradigm.

5.3.7 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 and 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 [105] was conceived expressly to support electronic commerce. For this reason, the term "mobile agent" is often related with electronic commerce. Other applications of code mobility to electronic commerce can be found in [61, 100].

5.3.8 Autonomous agents

Autonomous agents are applications that are able to achieve a goal, reacting to events in a dynamic environment and performing actions on the environment [59]. Autonomous agents are directed by some knowledge of the relationships among events, actions, and goals. Such knowledge can be exchanged with other agents, or increased by some inferential activity [31]. Autonomous agents are often required to be *mobile*, i.e., to be able to move across environments. The mobile code framework, providing the abstractions of locality and migrating components and the technologies supporting these abstractions, has been considered as a suitable platform for autonomous agents applications. For this reason, sometimes autonomous agents applications have been even erroneously identified with the concept of code mobility itself.

Chapter 6

A Case Study

6.1 Introduction

The purpose of this section is to illustrate how the taxonomy we presented in chapters 3 – 5 can be used to guide the Software Engineer through the design and implementation phases of the development process of a mobile code application. To this end, we use a case study, i.e., the design and implementation of a distributed information retrieval system. The case study is used to:

- highlight when the concepts discussed so far come into play during the development process;
- provide guidelines for the selection of mobile code paradigms and technologies;
- discuss in detail the relationships between paradigms and technologies.

We envision an idealized process by which software designers are equipped with methods that allow them to select the most appropriate design paradigms for any specific application. The software architecture designed according to the selected design paradigms should then be mapped onto an implementation using the technology (languages and their support tools) that best fit the chosen paradigms.

The suggested development process, guided by our taxonomy, is 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 in Section 6.2 on the basis of the discussion at the beginning of Chapter 5. The second step involves identifying the suitable paradigms for the design of the application at hand. This is done informally and qualitatively in Section 6.3. 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 6.4 we propose that a model of the application functionality should be built to enable quantitative analysis of the tradeoffs. Finally, after the suitable paradigms have been chosen, the technology for implementation has to be selected by examining the tradeoffs highlighted in Section 4.3.6, e.g., trading ease of programming for lightweight implementation.

In sections 6.5– 6.6 we exemplify these relationships by implementing several designs of the case study application using different types of technologies.

6.2 The problem: minimizing network traffic

The fast growth of the Internet [34] and the success experienced by the World Wide Web infrastructure [7] have changed the way information is accessed and distributed. Centralized information systems have evolved into a global distributed information repository interwoven with cross-references. The systems that participate in the World Wide Web are managed in a decentralized, autonomous way. While this evolution has provided a larger number of users with access to information and services, at the same time it has made some tasks considerably more complex. A notable example is represented by the retrieval of information concerning a particular subject [14].

Several systems, called *robots*, *spiders*, *wanderers*, *worms*, or *internet agents*, have been implemented to automate the process of finding and indexing the information provided by the World Wide Web infrastructure [54]. These systems contact WWW servers, retrieve their pages and analyze them looking for keywords and references to other locations. This process is repeated recursively. This way, *robots* build an indexed map of the available information. Users perform queries on these huge indexes in order to obtain links to the information sources that could satisfy their requests.

The main problem of this approach is that it produces a lot of traffic. For each indexing, a robot may contact thousands of sites, downloading through the network documents that maybe will never be accessed. In addition, since the World Wide Web by its very nature is subject to frequent change, information indexing must be performed on a regular basis. In some cases, this has lead to exhaustion of the available bandwidth in subnetworks hosting WWW servers frequently accessed by robots. Even though protocols to limit the scope of information indexing have been designed [53] and some guidelines for traffic-wise development of robots have been developed [52], the client-server approach to information retrieval shows its limits, at least as far as network traffic is concerned.

The mobile code approach has been proposed as a way to reduce the network traffic generated by information retrieval over a set of information servers. In particular, it is often stated that if this kind of application would be designed with "agents" interacting with servers, performance could improve and network traffic would be reduced. This claim, partially supported by intuition, has not yet been precisely assessed. This is partly due to the lack of actual applications that may act as testbeds for the new approach. In the following, we analyze the problem of optimizing network traffic in distributed information retrieval, in order to find if this claim has an actual justification.

Our study application is composed of several *document servers* and of a *search engine*. Servers are distributed over a set of computational environments (CEs). Servers are identified by a location, i.e., the computational environment they are running on, and a symbolic name that uniquely identifies a particular server in the corresponding CE. Each server manages a set of documents, that, for the sake of simplicity, we suppose identified by an integer number. Each document is composed of a *header* containing the document's keywords and

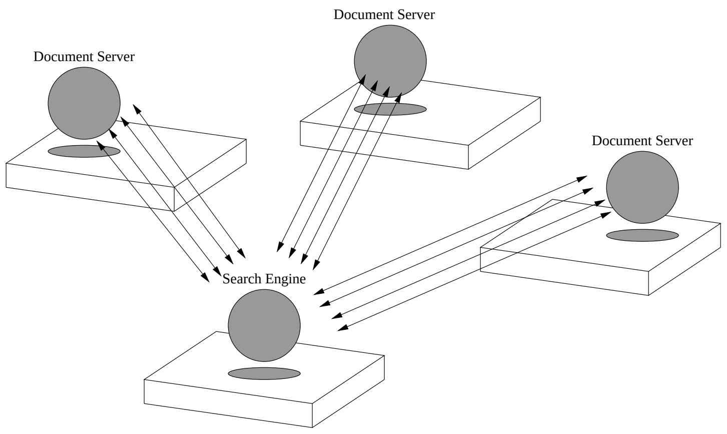

Figure 6.1: Client-Server design.

other meta-information, and a *body* containing the document's data and possibly references to other documents. The search engine's task is to gather the largest amount of information associated with a particular keyword, given an initial set of servers. If during the query of these initial servers some references to other servers are found, the search engine continues the search on those servers.

6.3 Designing the application

Usually, an application (or parts thereof) may be designed following different paradigms. In our case study, we have designed our application using the well-known Client-Server (CS) paradigm and two mobile code paradigms, namely, Remote Evaluation (REV) and Mobile Agent (MA). Note that, in doing this, we ruled out the Code On Demand (COD) paradigm which, although useful for augmenting the know-how of a component, in this case does not help us in trying to optimize communication among the data mining application and the servers.

Client-Server design. According to the CS paradigm, document servers play the role of service providers while the search engine is a single component that acts as a client. Each server offers a *header* service that takes as parameter an integer n that identifies a document and returns the header of the n^{th} document, or an error if such document does not exist. In addition, the server exports a *body* service that takes as parameter an integer n that identifies a document and returns the body of the selected document.

The search engine queries the servers contained in its initial set. For each server it requests all the document headers. If an header contains the keyword the search engine is looking for, then the corresponding document body is downloaded by invoking the *body* service. If the document body contains references to other servers, they are merged into the initial list. This process goes on until there are no more unqueried servers in the list. A high-level view of the components of the CS architecture and their interactions is given in Figure 6.1.

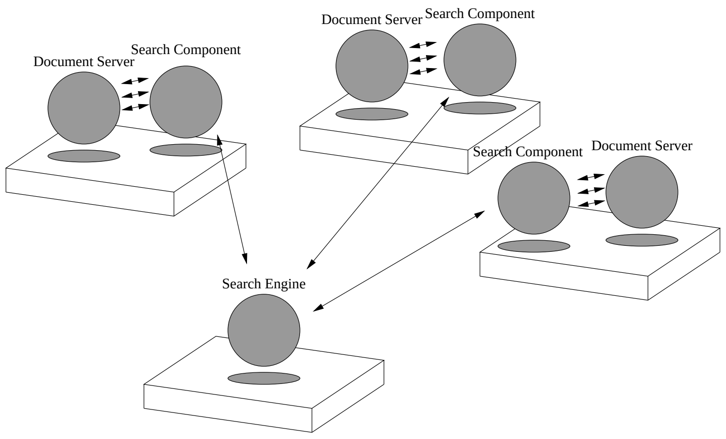

Figure 6.2: Remote Evaluation design.

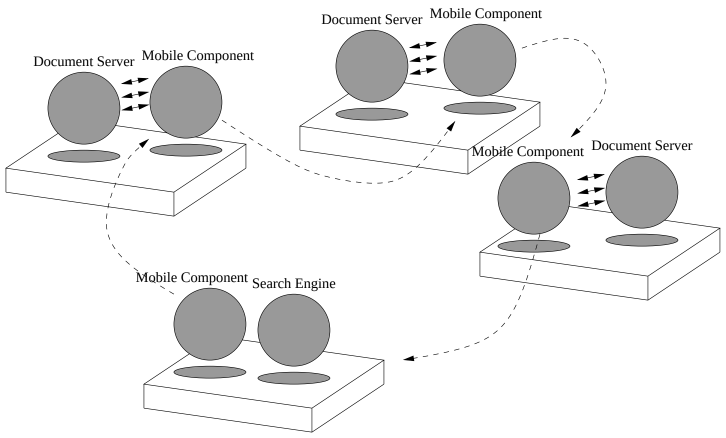

Figure 6.3: Mobile Agent design.

Remote Evaluation design. According to the REV paradigm, the search engine component sends the code to perform the query to the location of each server. The code is evaluated on the remote locations where servers are located, i.e., new components are created to execute the code on the destination CEs. These components query the server located at the same CE using the same primitives that servers export in the CS case. When a remote component terminates its task, it sends back to the search engine the results of the search. If the results contain some references to other servers, new REV interactions with these locations are started. The REV-based design of the application is sketched in Figure 6.2.

Mobile Agent design. Following the MA paradigm, the search engine functionality is modeled as a component that roams through the locations of the servers in order to collect information about the given keyword. The mobile component is provided with the initial server list. It moves to the location of the first server and performs the search on the local server. If relevant documents are found, its server list is updated accordingly. Then, it moves to the CE of the following server, bringing with itself the results of the previous query. Eventually, it returns to the originating site and delivers the results. The design exploiting the Mobile Agent paradigm is presented in Figure 6.3.

6.4 Choosing the right paradigm

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

There is no paradigm that is better than others in absolute terms. In particular, the mobile code paradigms do not necessarily prove to be better than the traditional ones. The choice of the paradigm must be performed on a case-by-case basis, according to the type of application and the particular functionality to be designed. For instance, the best paradigm for a network management application could be different from the paradigm that is best suited for an electronic commerce application. In addition, different functionalities may be conveniently designed following different paradigms.

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

6.4.1 Description of the model

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

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

For simplicity, we make the following additional assumptions:

- all requests sent to the document servers have a fixed length (r),
- each server manages the same number D of documents,
- the relevant information is uniformly distributed among a set of N sites, being i the ratio between relevant and total documents,

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

Table 6.1: Parameters for the model of a simple information retrieval application.

- documents have constant length. h and b are the size of the header and the body, respectively.

The application is characterized by the parameters shown in Table 6.1.

6.4.2 Evaluating the paradigms

Client-Server. If we design the application using the Client-Server paradigm, we have a search engine component on one site that will interact remotely with the N document servers. For each site, the search engine issues D requests for document headers and $i \times D$ requests for document bodies. Thus, the total traffic is:

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

Remote Evaluation. Using the REV paradigm, the search engine can obtain the execution of the filtering task on the same site that holds the document server. Thus, for each site H, the search engine sends a request to an executor component on H containing the code of the filter. C_{REV} is the size of this code. The executor component performs the requests to the document server and sends the relevant document bodies across the network back to the search engine. The engine updates the "see also" list on the basis of the documents contents and then focuses on another site.

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

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

Mobile Agent. In the MA paradigm, a mobile component migrates on each interesting site, performs all the interactions with the document server and the filtering locally, and saves in its state all the relevant information and the "see also" list.

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

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

that is,

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

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

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

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

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

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

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

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

This model could be extended in the following directions:

- *non-uniform networks*: we may assign different costs to different links in the network. This way, we may simulate disconnected/mobile computing, or networks characterized by clusters of sites with high-speed connections among sites within a cluster, and unstable and slow links among clusters.

- *security/CPU costs*: we can assign a non-null cost to the execution of code on remote hosts. This cost should include a fixed part that accounts for security procedures, e.g., authentication or code analysis, and a variable part proportional to the CPU usage.

- *memory usage*: we may assign a cost proportional to the amount of memory used by the executor of the code.

In addition, it is worth noting that small changes in the functionality can modify slightly the tradeoff. For example, if the search engine has to find the single document that contains in its body the highest number of occurrences of the given keyword, the expressions for the traffic in the three cases are the following.

In the case of the CS paradigm we have:

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

The expression is identical to the one deduced for the original functionality, because in order to determine the number of occurrences of the keyword in the body of the documents, all the relevant ones must be downloaded.

If we use the REV paradigm, this check can be performed at the document server's site and therefore, for each server, only the document with the higher number of occurrences will be returned as the result. As a consequence the expression for the traffic will be:

$$T_{REV} = (r + C_{REV} + b)N,$$

that is totally independent of the size of the document database.

The expression of the MA-based design becomes:

$$T_{MA} = (r + C_{MA} + \bar{s} + b)(N + 1),$$

which is linear with respect to the number of sites.

Thus, in this case the CS paradigm is definitely not convenient. The REV paradigm is the one that generates the minimum network traffic. On the other hand, since the difference between T_{REV} and T_{MA} is not relevant as in the expressions obtained for the original functionality, one may choose to adopt the MA paradigm on the basis of other benefits, e.g., the ability to perform the functionality when the search engine site is disconnected from the network.

6.5 Implementing the application

Having designed an application according to some paradigm, one has to choose a technology to implement it. Given a particular paradigm, which technology should be used?

As far as mobility is concerned, we can distinguish implementation technologies in three broad classes (see Section 3.3):

Non-mobile These technologies enable the communication between remote executing units (EUs) in the form of message exchange but they do not provide any form of code mobility. A typical example is RPC [11].

Weakly mobile These technologies provide mechanisms that enable an EU to send code to be executed in a remote computational environment (CE) together with some initialization data or to fetch code from a remote CE.

Strongly mobile These technologies enable EUs to move with their code and execution state to a different CE.

A first discussion of the relationships between paradigms and technologies has been given in Section 4.3.6. Here, we implement the different designs of our application using different technologies belonging to the classes listed above in order to show in greater detail which are the advantages and disadvantages of each solution. For the actual implementation we use Tcl-DP [86], Obliq [16], and Agent Tcl [36].

Tcl-DP is an extension of the Tcl language [69] for distributed programming. Tcl-DP provides support for TCP/IP and RPC programming. It is a non-mobile technology.

Obliq is an untyped, object-based, lexically scoped, interpreted language. It supports remote method invocation and remote evaluation of code. Therefore it is a weakly mobile technology.

Agent Tcl provides a Tcl interpreter extended with support for EU migration. An executing Tcl script can move from one host to another with a single `jump` instruction. A `jump` freezes the program execution context and transmits it to a different host which resumes the script execution from the instruction that follows the `jump`. Therefore, Agent Tcl is a strongly mobile technology.

Sections 6.5.1 through 6.5.3 provide an informal and succinct description of how the designs based on the CS, REV, and MA paradigms can be implemented using Tcl-DP, Obliq, and Agent Tcl. A complete description of the prototypes can be found in [24].

6.5.1 Implementing the Client-Server architecture

Non mobile technology. In this version of the application, we use Tcl-DP to implement the CS architecture. The document server component is implemented as a process running a Tcl-DP interpreter acting as an RPC server that exports the *header* and *body* services with the characteristics described in Section 6.2. The search engine component is implemented as another process running a Tcl-DP interpreter that acts as an RPC client. The client queries the server using RPC primitives. The CEs are represented by UNIX hosts.

Weakly mobile technology. We use the Obliq language to implement this version of the application architecture. Even if Obliq enables the remote invocation of methods, we use the mechanisms for remote evaluation of code offered by the language, since our goal here is to evaluate how a technology based on the remote evaluation of code supports the CS paradigm. We implement locations as Obliq interpreters that export *engines* (i.e., execution services), and document servers as objects that enclose the document database data and provide the *header* and *body* methods. The search engine is a thread in an interpreter that uses the *engines* exported by other interpreters in order to evaluate segments of code remotely. At run-time, the search engine chooses a server and sends a piece of code containing a single method invocation[1] to the remote engine of the interpreter containing the document server. The single invocation is performed in the remote interpreter by a newly created thread that, subsequently, delivers back the results.

Strongly mobile technology. We use Agent Tcl to implement the Client-Server architecture. In this setting, the document servers are stationary agents that accept queries from agents located on the same CE (represented by a UNIX host extended with the Agent Tcl run-time support). The search engine is a moving agent that, in order to query a document server, jumps to the corresponding CE, performs a single query, and then jumps back to its starting CE.

6.5.2 Implementing the Remote Evaluation architecture

Non mobile technology. We use Tcl-DP to implement the REV architecture. In this case, the remote CE is extended with a process running the Tcl-DP interpreter that acts as a code executor. This component is an RPC server that exports the service *execute*. This service takes as a parameter a Tcl-DP script. Servers are Tcl-DP interpreters that export the *header* and *body* services as in the CS case, but only to local processes. When the search engine wants to evaluate remotely the code that performs a set of queries, it sends to the remote executor a service request containing the query script. In turn, the executor interprets the script that queries the local document server and then returns the results back.

Weakly mobile technology. We use Obliq to implement the REV architecture. In this implementation, the search engine thread requests the execution of the query code to the remote engine corresponding to a particular document server. A new thread is created to execute the code. The newly created thread performs all the needed queries to the local server, which, as in the case of CS, is an object owned by the remote interpreter. Then, the results are delivered back to the source site.

Strongly mobile technology. We use Agent Tcl to implement the REV architecture. In this implementation, for each document server the search engine creates an agent that jumps to the site of the server, performs the queries on

[1]Note that if the code would contain several invocations of the services exported by the document server we would violate the structure imposed by the Client-Server design.

the local document server, and then returns to the original site with the results. Such results are then analyzed and possibly the "see-also" list is updated.

6.5.3 Implementing the Mobile Agent architecture

Non mobile technology. We use Tcl-DP to implement the MA architecture. In this case, the document servers are RPC servers that accept queries from local processes while the search engine's mobile component is a Tcl-DP script. In order to move from site to site, the mobile component packs its code and state (i.e., its "see-also" list, the values of some state variables that keep trace of its computation, and the documents retrieved so far) into a message and then sends the message to an executor that unpacks the message and creates a new component. The new component uses the information stored in the state part of the message in order to restore its execution flow. After dispatching the message to the remote executor, the sender terminates.

Weakly mobile technology. We use Obliq to implement the MA architecture. Since Obliq threads cannot migrate from one interpreter to another, when the search engine's mobile component must migrate to another interpreter it creates a copy of itself on the remote site and then terminates. In order to keep track of the execution, an object is used to maintain the state of the component. After each "jump", the newly created thread must create a local copy of the object representing the state and perform some operations based on its contents in order to restore the state of the computation.

Strongly mobile technology. We use Agent Tcl to implement the MA architecture. The search engine's mobile component is an Agent Tcl agent. At startup, the agent migrates to the location of the first document server in the given list. There, it queries the local document server and, if references to other document servers are found, it updates its "see-also" list. Then it migrates to the CE of the next server. This process is repeated until its "see-also" list becomes empty. Eventually, the agent jumps back to its starting location and reports its findings.

6.6 Discussion

The implementations described in the previous sections show that paradigms and technologies are not completely orthogonal. In principle, it is possible to implement applications developed with any paradigm by using any kind of technology, given that such technologies allow for the communication between EUs. However, we have found that some technologies are more suitable to implement applications designed using particular paradigms. Unsuitable technologies force the developer to program, at the application level, some mobility mechanisms or force an inefficient, counter-intuitive use of the existing ones.

As shown in Table 6.2, non mobile technologies are well suited for implementing architectures based on the CS paradigm. If they are used to implement REV-based architectures, they force the implementor to use (unnaturally) code as data and to program the evaluation of such code explicitly. Even worse, if non-mobile technologies are used to implement MA-based architectures, the

Technologies	Paradigms CS	REV	MA
Non mobile	Well suited	Code as data Program interpretation	Code and state as data Program state restoring Program interpretation
Weakly mobile	Code is a single instruction Creates unnecessary execution units	Well suited	State as data Program state restoring
Strongly mobile	Code is a single instruction Creates unnecessary execution units Move state back and forth	Manage migration Move state back and forth	Well suited

Table 6.2: Relationships among paradigms and technologies.

programmer has also to manage explicitly state marshaling and unmarshaling, i.e., auxiliary variables must be used to keep the state of the computation and unnatural code structures must be used to restore the state of a component after migration to a different CE.

Weakly mobile technologies that allow to execute segments of code remotely are naturally oriented towards the implementation of applications designed according to the REV paradigm. These technologies are inefficient to implement CS architectures since they force the remote execution of segments of code composed of a single instruction. Therefore a new EU is created in order to execute this "degenerate" code. On the contrary, in order to implement applications based on the MA paradigm, the programmer has to manage, at the program level, the packing/unpacking of the variables representing the state and the restoring of the EU execution flow [2].

Strongly mobile technologies are oriented towards MA-based applications while they are not suited for implementing applications based on the CS and REV paradigms. In the former case, the programmer has to "overcode" an agent in order to have it moved to the server site, execute a single operation and jump back with the results. Such implementations could be rather inefficient since the whole EU state is transmitted back and forth across the network. In the latter case, in addition to the code to be executed remotely, the implementor has to add the migration procedures. Furthermore, the state of the EU is to be transmitted over the network.

Summing up, technologies may reveal to be too powerful or too limited to implement a particular architecture. In the first case resources are wasted, resulting in inefficiency. In the second case, the programmer has to code all the mechanisms and policies that the technology does not provide.

[2]This is a very common situation. In fact, most existing mobile code technologies are weakly mobile, because of the easier implementation of weak mobility mechanism. Nonetheless, practitioners tend to think in terms of the MA paradigm. As a consequence, there are many examples of mobile code applications that use awkward conditional structures to restore, at the logical level, the computational flow after migration.

6.7 Concluding remarks

This case study exemplifies the development process of a mobile code applications. We have shown how the application-level benefits provided by the mobile code approach (see Chapter 5) can be used in the requirements analysis phase of the development process to evaluate if the use of mobile code is convenient. Then, we showed how the mobile code paradigms introduced in Chapter 4 can be used as building blocks to define the application architecture during the design phase. Eventually, we have shown how technologies to be used in the implementation phase can be selected on the basis of the characterization presented in Chapter 3.

We did not cover phases of the development process other than requirements analysis, design, and implementation. As an example, testing and debugging phases are strongly influenced by the introduction of mobility. Testing of mobile code applications is much more difficult than testing of "traditional" distributed applications since the configuration of components and the binding among components and code is dynamic. In addition, debugging code that is executed remotely is a challenging task. Further work is needed to fully understand the effects of the new computational infrastructure provided by the mobile code approach on the entire life cycle of mobile code applications.

The proposed case study is different from the development of real-life mobile code applications in different ways. First of all, usually a single design for the application is chosen after the evaluation of the available alternatives, as described in Section 6.4. Second, usually a single technology is chosen for the implementation. We developed nine different prototypes in order to experiment different compositions of paradigms and technologies, but this is not the process that should be followed for an actual application. Third, as described at the end of Section 6.4, *all* the functionalities of an application should be analyzed and assessed. Taking into account the requirements of the different functionalities may change the way technologies are selected. In fact, the choice of the technology used to implement the application must take into account not only which are the mechanisms that better support a given functionality designed following a particular design paradigm, but also how the technology fits the global application development. For example, let us suppose that we have a functionality that is conveniently designed following an REV paradigm, and another one that could benefit from an MA-based design. Now suppose that we are faced with the choice between a strongly mobile code system which does not provide good support for stand-alone code shipping and a weakly mobile technology which provides it. If the second functionality is to be used less frequently than the first one, we may want to sacrifice the straightforward implementation achievable with the strongly mobile technology and use the weakly mobile one, to obtain better support in the key functionalities and keep the uniformity of the development tools.

Chapter 7

Protecting Mobile Code

7.1 Introduction

While the mobile code approach to designing and implementing distributed application has proved to be valuable, it raises some serious security issues, as described in Section 3.6. By far the hardest security problem is represented by the protection of executing units (EUs) from malicious actions of the hosting computational environments (CEs).

We propose a system that allows for detecting any possible misbehavior of the CE with respect to a roaming EU by using cryptographic traces. Traces are logs of the operations performed by a roaming component during its lifetime. Our system allows an EU owner to check with a high degree of confidence, after EU termination, if the execution history of the EU conforms to a correct execution.

7.2 Agents and security

The problem of protecting a mobile component from the hosting computational environment has been addressed in different ways.

Most mobile code systems consider the site as a trusted entity and therefore they do not provide any mechanism to protect mobile EUs execution. A different approach has been followed by some researchers that have designed security mechanisms for the protection of roaming components. These same researchers have tried to identify which goals are impossible or very difficult to achieve [20, 27]. Tasks being considered impossible are: prevention of tampering with EUs state (unless some kind of tamper-proof hardware is adopted), guarantee that an environment will execute an EU correctly and to its completion, guarantee that the EU is moved to the requested CEs, and total protection of EU's data from disclosure[1].

In [26] a protection mechanism against sites trying to tamper with EUs state is presented. The mechanism is based on state appraisal functions. These functions express invariants that the EU state must satisfy. This way, some malicious attempt to tamper with the EU state can be detected.

[1]Partially encrypted EUs are a possible partial solution to the problem.

Our approach is stronger: we want to be able to detect any possible unauthorized modification of a mobile EU code and state. Obviously, one cannot foresee the information that an EU will receive from the CE it will visit or from other EUs it will interact with. From this point of view, the application developer must use the same caution required when developing privileged programs that could receive parameters by untrusted principals (e.g., CGI scripts).

7.3 Cryptographic traces

As stated in [20], it is impossible to *prevent* malicious or faulty sites from tampering with EUs. Sites must have access to an EU code and state in order to support its execution. Still, there are means to detect abnormal behaviors or unauthorized modifications of EU components.

We propose a mechanism for tracing the execution of a migrating EU in an unforgeable way. Such traces can be used as a basis for program execution verification, i.e., for checking the EU code against a supposed history of its execution. This way, cryptographic tracing allows the EU owner to prove, in case of tampering, that the operations the EU is accounted for could have never been performed.

The proposed protocol assumes that all the involved principals, namely users and site owners, own a public and a secret key that can be used for encryption and digital signatures [78]. The public key of a principal A is denoted by A_p, while A_s is used for the corresponding secret key. The principals are users of a *public key infrastructure* [45] that guarantees the association of a principal with the corresponding public key. We assume that, at any moment, any principal can retrieve the public key of any other principal and verify its integrity. For the sake of simplicity, both the public key of a user and the associated certificate will be denoted by A_p. The process of encrypting a message m with a key is expressed by $A_x(m)$. In addition, we will use one-way hash functions in order to produce cryptographically secure compact representations of messages. The hash value obtained by application of the function H to the message m is denoted by $H(m)$. Several examples of public key cryptosystems[2] and one-way hash functions can be found in [84].

A moving EU is composed by some code C and some state S^i that has been determined, at some specified point i, by code execution.

We will assume that the code is *static* with respect to the lifetime of the EU and that it is composed of a sequence of statements. Statements can be of two types. A *white* statement is an operation that modifies the content of the program state only on the basis of variables internal to the program. For example, the statement $x := y + z$, where x, y, and z are variables of the program, is a white statement. A *black* statement modifies the state of the program using information received from the external execution environment. For example, the statement $read(x)$, that assigns to the variable x a value read from the terminal, is a black statement.

[2]Public-key cryptography is slow when compared to symmetric cryptography. In the sequel, when we will need bulk encryption using public keys we will assume that the message has been encrypted with a randomly generated secret key and that the key has been protected with the original public key. I.e., if the application of a symmetric encryption process, parameterized by a key K to a message m is represented by $K(m)$, then $A_x(m)$ is equivalent to $A_x(K), K(m)$.

We assume that all the interpreter implementations are certified to be correct and to respect the semantics of the language[3].

A trace T_C of the execution of program C is composed of a sequence of pairs $\langle n, s \rangle$, where n represents a unique identifier of a statement, and s is a *signature*. When associated to black statements, the signature contains the new values assumed by internal variables as a consequence of statement execution. For example, if the $read(x)$ instruction gets the integer 3 from the terminal, the associated signature will be $x := 3$. The signature is empty for white statements.

In the following sections we will address problems of increasing complexity. In particular, we will follow the distinction made by [68], firstly tackling the problem of protecting *remote code execution*, and then analyzing *boomerang agents* and *multi-hop agents.*

7.3.1 Remote code execution

Suppose that user A wants to execute code C on site B. Then, A sends B the following message :

$$A \xrightarrow{m_0} B : A, B_p(C), A_s(H(C), B, A', t_A, i_A).$$

Since the code is protected using the public key of B, it is accessible by B only. When B receives the messages, it decrypts the second part of the message using its secret key B_s to obtain the code. Then it uses the principal name specified in the first part of the message (A) to retrieve the corresponding public key A_p. The key is used in order to decrypt the third part of the message. B computes $H(C)$ and compares the result with the hash value contained in the third part of the message. If the two values match, B is assured that the message was sent by A, and that the code is unmodified. Since the third part of the message contains B, B can be sure that the message was intended for himself. The principal identified by A' is the recipient of every receipt token that will be produced by the protocol. A' and A may coincide, or may denote a different recipient. This option can be used in case A has disconnected after sending m_0 or if it requires that some different trusted third party collects the receipts (e.g., a digital notary). t_A represents a timestamp to guarantee freshness[4] and i_A represents a unique identifier to protect from replay attacks.

After authentication has been performed, B sends A' a receipt of the received message, containing the identity of the sender and the third part of message m_0, signed with his own secret key:

$$B \xrightarrow{m_1} A' : B, B_s(A, A_s(H(C), B, A', t_A, i_A)).$$

This way, A' can verify that B received a code execution request from A at time t_A and that A' was actually indicated as the recipient of receipt messages. At any moment, A can prove that B received C by showing to A' that the computation of the hash $H(C)$ produces a result corresponding to the value contained in m_1.

[3]For example, the interpreter owner must provide some kind of third-party certification of correct implementation of the language in order to be able to charge EUs for the services they use.

[4]The timestamp may be associated to a time interval in order to limit EU execution.

Once the initial handshake has been performed, B executes C, and produces a corresponding trace T_C. When the execution terminates, B sends a signed message to A' containing a checksum of the program final state S^1, a checksum of the execution trace T_C (whose extended form is stored–for a limited amount of time–by B), and the unique identifier i_A:

$$B \xrightarrow{m_2} A' : B, B_s(H(S^1), H(T_C), i_A).$$

Then, B sends a signed message to A containing the program final state S^1 (i.e., the results of the computation[5]) protected using A's public key together with the unique identifier i_A:

$$B \xrightarrow{m_3} A : B, B_s(A_p(S^1), i_A).$$

Since both the messages are signed, B cannot send a modified state to A or a modified checksum to A' without being detected.

If A suspects that B cheated while executing C, it can ask B to produce the trace and A' to produce the receipt messages. Then, A checks whether the trace is the supposed one by using the value $H(T_C)$ contained in m_2. Finally, A replicates the execution of the program C following the trace T_C. The validation process *should* produce the final state S^1. If it is not so, B cheated by modifying the code or by modifying some program variables, and B's misbehavior can be proved to third parties. Note that the complexity of the validation process is linear with the size of the execution trace.

7.3.2 Boomerang agents

A *boomerang agent* is an EU that at a certain point migrates to a remote site B in order to perform some task, and when the task is accomplished gets back to its home site.

In this case, there is some state S^0 associated to the code C. Therefore, the first message will be :

$$A \xrightarrow{m_0} B : A, B_p(C, S^0), A_s(H(C), H(S^0), B, A', t_A, i_A).$$

From the contents of the message B can be sure that it was the intended recipient and that both the code and state were actually sent by A and are untampered. Therefore, B sends A' the following receipt :

$$B \xrightarrow{m_1} A' : B, B_s(A, A_s(H(C), H(S^0), B, A', t_A, i_A)),$$

so that A' can verify that B received an EU to be executed from A at time t_A, and that A' was actually indicated by A as the recipient of receipt messages. At any moment, A can prove that the code and the state received by B were those specified in m_1 by a process analogous to the one performed in the previous case.

Then, B creates an EU image initialized with the code and the state provided by A and it resumes its execution. The execution goes on, generating the corresponding trace, until the EU decides to migrate back home. Then, B sends

[5]The state that is returned is the *final* state containing just the data that represent the outcomes of the computation.

A' a signed message containing an hash of the EU state just after the migration statement (S^1), a hash of the execution trace, and the unique identifier i_A:

$$B \xrightarrow{m_2} A' : B, B_s(H(S^1), H(T_C), i_A),$$

and, in addition, a signed message to A containing the program final state S^1 protected using A's public key together with the unique identifier i_A.

$$B \xrightarrow{m_3} A : B, B_s(A_p(S^1), i_A),$$

Since A owns the EU's code C and the current state, it can restore the computation of the EU.

Like in the previous case, if A suspects that B cheated, it can ask B to produce the trace. The validation process should produce the final state S^1 starting from state S^0.

7.3.3 Multi-hop agent

A *multi-hop agent* is an EU that, starts from a initial home site and then jumps from site to site to perform a particular task. The EU gets back home when the task has been accomplished. There are several problems related to the authentication of an EU that has visited a possibly large list of sites with different levels of trust. Here, we will tackle the problem of assuring that none of the visited sites can tamper with the roaming EU execution without being detected.

The first hop of the EU produces two messages similar to the ones produced in the previous case:

$$A \xrightarrow{m_0} B : A, B_p(C, S^0), A_s(H(C), H(S^0), B, A', t_A, i_A),$$

and:

$$B \xrightarrow{m_1} A' : B, B_s(A, A_s(H(C), H(S^0), B, A', t_A, i_A)).$$

Then B executes the EU until it requires to move to another site D. As a consequence, B stops the EU execution and sends A' a signed message containing the name of the site that will represent the next hop in the EU route, a hash value of the final state S^1, a hash value of the execution trace, and the unique identifier i_A:

$$B \xrightarrow{m_2} A' : B, B_s(D, H(S^1), H(T_C^1), i_A).$$

Then, B sends D a message composed as follows. Firstly there are the name of the sender and the name of the principal that is responsible for EU execution. The following part contains the unit code and the current state, encrypted with D's public key in order to protect the code and the state from unauthorized disclosure. Then there is a hash value of the current state and the name of the intended recipient signed with the sender secret key. The last part of the message contains a copy of the third part of m_0:

$$B \xrightarrow{m_3} D : B, A, D_p(C, S^1), B_s(H(S^1), D), A_s(H(C), H(S^0), B, A', t_A, i_A).$$

D decrypts the third part using its secret key D_s. Afterwards, D retrieves the public keys associated to principals A and B, and uses them to decrypt the remaining parts of the message. From the contents of the part signed by B, D

agent	append	array	break	case
catch	clock	concat	continue	error
expr	for	foreach	format	global
go	if	incr	join	lappend
lindex	linsert	list	llength	lower
lrange	lreplace	lsearch	lsort	pid
proc	reply	request	return	scan
service	set	split	string	subst
switch	unset	uplevel	upvar	while

Table 7.1: SALTA command set.

can argue that it was the intended recipient of the message, that the message was actually sent by B and that the state S^1 included in the third part was correctly transmitted. From the part of the message signed by A, D can deduce that the code contained in the third part is correct and that the EU was initially dispatched by A . In addition, D can be sure that A meant to use A' as recipient of the receipt messages.

Then, D sends A' a signed message containing a signed receipt of the last two parts of the message:

$$D \xrightarrow{m_4} A' : D, D_s(B_s(H(S^1), D), A_s(H(C), H(S^0), B, A', t_A, i_A)).$$

Now A' can be sure that D received C and S^1 correctly, i.e., B did not send D code or state that are different from those used in order to compute the hash values that have been delivered to A' in message m_2.

This protocol is repeated until the EU decides to terminate. In this case, the final site, say Z, sends A' a signed message containing the hash value of both the final state S^n and the execution trace T_C^n, and the unique identifier i_A:

$$Z \xrightarrow{m_n} A' : Z, Z_s(H(S^n), H(T_C^n), i_A).$$

Then, Z sends A a signed message containing the program final state S^n (i.e., the results of the computation) protected using A's public key, together with the unique identifier i_A:

$$Z \xrightarrow{m_{n+1}} A : Z, Z_s(A_p(S^n), i_A).$$

After having examined the computation results, if A thinks that one or more of the sites involved cheated, it can ask A' to provide the receipts generated by the EU trip. In addition, A asks each site to produce the corresponding traces. A simulates the EU execution starting from S^0 and following, in the right order, the execution traces. At each step i, the partial state S^i should produce a hash value equal to the one contained in the corresponding receipt message, otherwise the site at step i cheated.

```
1     : set home home.sweet-home.com
2     : go agents.virtualmall.com
3     : set shoplist [request directory query homevideo]
4     : foreach shop $shoplist {
4.1   :     go $shop
4.2   :     set price($shop) [request catalog movies "Pulp Fiction"]
      : }
5     : set best_price 21
6     : set best_shop none
7     : foreach p [array names price] {
7.1   :     if {$price($p) < $best_price} {
7.1.1:         set best_price $price($p)
7.1.2:         set best_shop $p
      :     }
      : }
8     : if {$best_price > 20} {
8.1   :     go home.sweet-home.com
8.2   :     request terminal print "No offers below \$20!"
8.3   :     exit
      : }
9     : go $best_shop
10    : set trans_id [request buymovie "Pulp Fiction" $best_price]
11    : go $home
12    : request terminal print "Transaction id: $trans_id"
13    : exit
      : }
```

Figure 7.1: The Pulp Fiction Agent.

`home`	`virtualmall`	`brock`	`towel`
1	3,shoplist = brock towel	4.2,price(brock) = 15	4.2,price(towel) = 17
2	4	4	4
12	4.1	4.1	5
13		10,trans_id = PF11	6
		11	7
			7.1
			7.1.1
			7.1.2
			7
			7.1
			7.1.1
			7.1.2
			7
			8
			9

We made the following substitutions:

`home`	=	`home.sweet-home.com`
`virtualmall`	=	`agents.virtualmall.com`
`brock`	=	`agents.brockbuster.com`
`towel`	=	`agents.towelrecords.com`

Figure 7.2: Agent execution traces.

7.4 The SALTA language

The SALTA language (*Secure Agent Language with Tracing of Actions*[6]) is a modified version of the Safe-Tcl language [13, 71]. The available Tcl [69] command set is showed in Table 7.1. The Safe-Tcl language has been restricted further, and some new instructions have been added: `request`, `service`, `reply` and `go`.

The `request` command takes as a parameter an agent name, a command name and possible parameters. The request to execute the command is dispatched to the specified agent (if it exists). The server agent will accept the request by using the `service` command which returns the command string that represents the command invocation. Upon completion, the agent returns the service results by using the `reply` command which delivers the results to the agent that made the service request.

Agents are identified by using an URL-like scheme, in which an agent is identified by the protocol modifier `agent`, followed by the name of the host (possibly extended with a port number) running the computational environment in which the agent resides, and the agent name. For example an agent named `luke` at host `agents.starwars.com` is referenced by `agent://agents.starwars.com/luke`.

The `go` command causes the suspension of the program, the packing of its code and execution state (values of variables, call stack, and program counter), and the shipping of the packed executing unit to a remote computational environment specified as a parameter of the `go` command. When the packed

[6]"Salta" is also an italian verb meaning "jump".

executing unit reaches its destination, it is unpacked, its state is restored and its execution is restarted from the command following the `go`. From this point of view SALTA is similar to Agent-Tcl [37].

7.5 A Pulp Fiction Agent

In order to give a precise idea of the operations involved in EUs execution tracing and verification, we describe a simple electronic commerce application.

A user, at site `home.sweet-home.com` wants to buy a home video of Tarantino's *Pulp Fiction* movie. Therefore, he dispatches an EU to a site called `agents.virtualmall.com` dedicated to maintain a directory of electronic shops. Once there, the EU performs a directory query for sites having home video offers. Then, the EU moves to each of the provided sites, where it contacts the local catalog agent in order to determine the current price of the *Pulp Fiction* home video. When all prices have been collected, the EU identifies the best offer and then, if the best price is less then a specified amount (say, twenty dollars), the EU goes to the selected site and contacts a selling agent to buy the home video. Finally, the EU goes back home and reports the transaction identifier associated to the purchase. Figure 7.1 shows the EU code.

Now, let us suppose that the directory service at `agents.virtualmall.com` has suggested two sites, namely `agents.brockbuster.com` and `agents.towelrecords.com`, and the prices of the *Pulp Fiction* home video are fifteen dollars at `agents.brockbuster.com` and seventeen dollars at `agents.towelrecords.com`. The execution trace will be the one shown in Figure 7.2.

Let us suppose that the Towel Records site wants to modify the Brockbuster offer pretending to be the cheapest offer for the movie. Then, before the EU computes the best price, the Towel Records site modifies the home video price associated to Brockbuster, raising its value to twenty-two dollars. As a consequence, the EU buys the home video at Towel Records and gets back home.

In order to verify EU execution, A retrieves the traces associated to the different sites visited by the EU, and the receipt messages associated to each hop. Then, A repeats the EU execution following the provided traces. When the verification process reaches instruction 7.1 (see figures 7.1 and 7.2) an inconsistency is flagged. Since `agents.brockbuster.com` sent a signed checksum of the EU state just before the EU left the site and `agents.towelrecords.com` has provided an identical checksum, it means that the modification must have happened at the Towel Records site. Thus, since there are no instructions that assign a value to the variable containing Brockbuster's price in those listed in the trace of the operations performed at Towel Records, A can prove that Towel Records cheated.

The same procedure can be used to flag out tampering with code, inconsistencies in state transmission, computational flow diversion, and so on.

7.6 Limitations of the approach

The proposed approach has some limitations.

First of all, the size of the traces may be huge, even if compressed. Several mechanisms can be used in order to reduce the size of the traces. For example, instead of a complete execution trace it could be possible to log just the points in execution where control flow changes (e.g., conditional statements and loops) and the signature of black statements. Another mechanism could be devised that allows the programmer to define a *range* of statements to be traced. The programmer may require that the values of some critical variables must satisfy a set of constraints before entering that particular group of statements. In a similar way, a mechanism could allow a programmer to specify that a group of statements must *not* be traced and that just the final values of the modified variables must be included in the trace. This mechanism could be useful in search procedures when, during a loop, several values are retrieved from the external environment and a small set of them is saved in the EU state.

Second, we made the assumption that EUs cannot share memory and are single threaded. If this is not the case, an extension to the tracing mechanism is required. In order to check the execution of an EU, a user would need the trace of all the EUs or threads that shared some memory portion with the EU thread under examination. In addition, traces should be extended with some timing information that allows for determining the order of the statements executed by the different threads. As one can easily understand, this mechanism would be practically unfeasible.

Third, the mechanism makes the assumption that the code is static. This forbids the use of optimization techniques like just-in-time compilation. In order to overcome this limit a just-in-time compiler should produce additional information (similar to the information used for debugging purposes) in the compiled module. This information should allow to map the execution of the compiled module on the source code of the EU.

Chapter 8

Conclusions

The impressive development of telecommunication, multimedia, and software technology has made it possible to envisage and create a wide range of new applications, for traditional and new domains of our society. However, this impressive development is presently limited by the relatively slow evolution of the computational infrastructure of computer networks, which is largely based on the client-server paradigm. Code mobility is a promising approach that aims at overcoming many of the drawbacks of the traditional approaches. Practitioners have been attracted by the potential of mobile code languages, but technologies are only one aspect of code mobility. Hence, developments in code mobility have not been supported by a comprehensive characterization and comparison of the proposed technologies and by a corresponding development of methodologies.

There are different kinds of confusions and misunderstandings on the meaning and role of many existing approaches. In some cases, terms are overloaded: the notion of code mobility itself can be interpreted in several and quite different ways. In other situations, we experience a lack of concepts and abstractions. In particular, there is quite often a confusion among the basic technologies for mobile code, the paradigms used during software design, and the applications developed using these technologies and paradigms. This situation hampers the understanding of the rationale and the real contribution of each specific approach. As a consequence, mobile code has not yet been widely accepted. This is not only due to the lack of methodologies and conceptual frameworks but also to the new security issues introduced by the mobile code approach. One of the most difficult problem to solve is the protection of roaming components from malicious computational environments.

In this thesis we proposed a set of concepts to progress towards a common vision of the issues and contributions in the area of mobile code. We identified three main classes of concepts: *applications*, *technologies*, and *paradigms*. Applications are the solutions to specific problems. Paradigms guide the design of applications. Technologies support their development. We surveyed each of these concepts and pointed out features, advantages, and disadvantages of existing approaches and proposals. We have analyzed the relationships and the mutual influences among these concepts from a development and management viewpoint. As one might expect, applications and paradigms are fairly orthogonal, i.e., an application can be designed following different paradigms for mobile code. This is not completely true for paradigms and technologies. Typically, technologies offer mechanisms that are oriented to support one or few paradigms. We also argue that the choice of a paradigm cannot be based on general and application-independent criteria: a variation in application requirements and constraints may suggest different implementation strategies. For instance, even if certainly elegant and appealing, the adoption of a mobile agent paradigm should be based on a careful evaluation of the operational profile. In turn, the implementation language should be selected taking into account the relationship between the features of the language and the paradigm(s) being selected. Once again, there is no clear winner or "silver bullet", and we definitely need to select and adopt a design and implementation strategy that fits the requirements and constraints of the application being developed.

Certainly, the framework presented in this thesis needs to be incrementally enriched and revised, taking into account experiences, results, and innovations that will be progressively achieved. We need to improve our classification by better analyzing the properties and weaknesses of the existing design paradigms. We also need to consolidate a conceptual framework for mobile code languages, that make it possible to compare them as we do with traditional programming languages. Finally, we need to further explore the relatively unknown world of applications and problems that can benefit from the adoption of technologies and methodologies based on the notion of code mobility. Nonetheless, we believe that the concepts presented in this thesis can be instrumental in the creation of a mature and comprehensive background for the evolution and further diffusion of mobile code applications and techniques.

In order to tackle the security problem, we conceived a new mechanism, based on execution tracing, that allows the owner of a mobile component to detect any possible attempt to tamper with component data, code, and execution flow, under certain assumptions. The proposed technique does not require dedicated tamper-proof hardware or trust between parties. A language that implements the system concepts has been described, together with a simple electronic commerce example.

The proposed security mechanism will be improved to include a transaction protocol that may be used to commit actions performed by agents. The transaction protocol could allow a site that represents an intermediate step in the agent route to verify code execution since agent start-up. We are working also on an extension of the protocol that guarantees privacy of the identities of the principals involved. In addition, the SALTA language is under development and much of our work will be dedicated to design a mobile code system that protects *both* executing units and computational environments.

Bibliography

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

[2] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In Vitek and Tschudin [101], pages 111–130.

[3] Y. Artsy and R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, pages 47–56, September 1989.

[4] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Experience with Distributed Programming in Orca. *IEEE Trans. on Software Engineering*, 18, March 1992.

[5] M. Baldi, S. Gai, and G.P. Picco. Exploiting Code Mobility in Decentralized and Flexible Network Management. In Rothermel and Popescu-Zeletin [81], pages 13–26.

[6] J. Baumann, F. Hohl, N. Radouniklis, K. Rothermel, and M. Straßer. Communication Concepts for Mobile Agent Systems. In Rothermel and Popescu-Zeletin [81], pages 123–135.

[7] T. Berners-Lee, R. Cailliau, H.F. Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, August 1994.

[8] K.A. Bharat and L. Cardelli. Migratory Applications. Technical Report 138, Digital Equipment Corporation, Systems Research Center, February 1996.

[9] S. Bhattacharjee, K.L. Calvert, and E.W. Zegura. An Architecture for Active Networking. In *High Performance Networking (HPN'97)*, April 1997.

[10] L. Bic, M. Fukuda, and M. Dillencourt. Distributed Computing Using Autonomous Objects. *IEEE Computer*, August 1996.

[11] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, 2(1):29–59, February 1984.

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

[13] N. Borenstein. EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail. Technical report, First Virtual Holdings, Inc, 1994.

[14] P.M.E. De Bra and R.D.J. Post. Information Retrieval in the World-Wide Web: Making Client-Based Searching Feasible. In *Proceedings of the First International World-Wide Web Conference*, Geneva, Switzerland, May 1994.

[15] T. Cai, P. Gloor, and S. Nog. DataFlow: A Workflow Management System on the Web Using Transportable Agents. Technical Report TR96-283, Dept. of Computer Science, Dartmouth College, Hanover, NH, 1996.

[16] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[17] N. Carriero and D. Gelernter. Linda in Context. *Comm. of the ACM*, 32, April 1989.

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

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

[20] D.M. Chess, B. Grosof, C.G. Harrison, D. Levine, C. Paris, and G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communication*, October 1995. Also available as IBM Technical Report.

[21] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In Vitek and Tschudin [101], pages 93–111.

[22] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems. In *Proc. of the 20th Int. Conf. on Software Engineering*, Kyoto, Japan, 1998.

[23] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software Practice and Experience*, 21(8):757–785, August 1991.

[24] T. Dovera and N. Nespoli. Paradigmi e tecnologie per lo sviluppo di applicazioni basate su codice mobile. Master's thesis, Politenico di Milano, 1996.

[25] J.R. Falcone. A Programmable Interface Language for Heterogeneous Distributed Systems. *ACM Trans. on Computer Systems*, 5(4):330–351, July 1987.

[26] W.M. Farmer, J.D. Guttman, and V. Swarup. Security for Mobile Agents: Authentication and State Appraisal. In Springer, editor, *Proc. of the 4th European Symp. on Research in Computer Security*, volume 1146 of *LNCS*, pages 118–130, Rome, Italy, September 1996.

[27] W.M. Farmer, J.D. Guttman, and V. Swarup. Security for Mobile Agents: Issues and Requirements. In *Proc. of the 19th National Information Systems Security Conf.*, pages 591–597, Baltimore, MD, USA, October 1996.

[28] G.H. Forman and J. Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4):38–47, 1994.

[29] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 1998. To appear.

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

[31] M. Genesereth and S. Ketchpel. Software Agents. *Comm. of the ACM*, 37(7), July 1994.

[32] C. Ghezzi and G. Vigna. Mobile Code Paradigms and Technologies: A Case Study. In Rothermel and Popescu-Zeletin [81], pages 39–49.

[33] G. Goldszmidt and Y. Yemini. Distributed Management by Delegation. In *Proc. of the 15th Int. Conf. on Distributed Computing*, June 1995.

[34] M. Gray. Growth of the World-Wide Web. `http:/www.mit.edu/~gray/-web-growth.html`, December 1993.

[35] R. Gray, G. Cybenko, D. Kotz, and D. Rus. D'agents: Security in a multiple-language, mobile agent system. In G. Vigna, editor, *Mobile Agent Security*, LNCS. Springer, 1998.

[36] R.S. Gray. Agent Tcl: A transportable agent system. In *Proc. of the CIKM Workshop on Intelligent Information Agents*, Baltimore, Md., December 1995.

[37] R.S. Gray. Agent Tcl: A flexible and secure mobile agent system. In *Proc. of the 4th Annual Tcl/Tk Workshop*, pages 9–23, Monterey, Cal., July 1996.

[38] R.S. Gray, D. Kotz, S. Nog, D. Rus, and G. Cybenko. Mobile agents for mobile computing. In *Proc. of the 2nd Aizu Int. Symp. on Parallel Algorithms/Architectures Synthesis*, Fukushima, Japan, March 1997.

[39] C.G. Harrison, D.M. Chess, and A. Kershenbaum. Mobile Agents: Are they a good idea? In Vitek and Tschudin [101], pages 25–47. Also available as IBM Technical Report.

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

[41] F. Hohl. An approach to Solve the Problem of Malicious Hosts in Mobile Agent Systems. Technical Report 1997/03, Universitat Stuttgart, Fakultat Informatik, 1997.

[42] F. Hohl. Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. In G. Vigna, editor, *Mobile Agents and Security*, LNCS. Springer, 1998.

[43] T. Imielinsky and B.R. Badrinath. Wireless Computing: Challenges in Data Management. *Comm. of the ACM*, 37(10):18–28, 1994.

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

[45] ITU-T. Information Technology - Open Systems Interconnection - The Directory: Authentication Framework. ITU-T Recommendation X.509, November 1993.

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

[47] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained Mobility in the Emerald System. *ACM Trans. on Computer Systems*, 6(2):109–133, February 1988.

[48] G. Karjoth, D. Lange, and M. Oshima. A Security Model for Aglets. *IEEE Internet Computing*, 1(4):68–77, July-August 1997.

[49] J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4):21–30, 1997.

[50] F.C. Knabe. *Language Support for Mobile Agents.* PhD thesis, Carnegie Mellon Univ., Pittsburgh, PA, USA, December 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.

[51] P. Knudsen. Comparing two Distributed Computing Paradigms - a Performance Case Study. Master's thesis, Univ. of Tromsø, 1995.

[52] M. Koster. Guidelines for Robot Writers. `http://info.webcrawler.-com/mak/projects/robots/guidelines.html`, 1993.

[53] M. Koster. A Standard for Robot Exclusion. `http://info.webcrawler.-com/mak/projects/robots/norobots.html`, 1994.

[54] M. Koster. Robots in the Web: threat or treat? *ConneXions*, 9(4), April 1995.

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

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

[57] R. Lea, C. Jacquemont, and E. Pillevesse. COOL: System Support for Distributed Object-Oriented Programming. *Comm. of the ACM*, 36(9):37–46, November 1993.

[58] 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, November 1996.

[59] P. Maes. Agents that Reduce Work and Information Overload. *Comm. of the ACM*, 37(7), July 1994.

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

[61] 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.

[62] Sun Microsystems. The Java Language: An Overview. Technical report, Sun Microsystems, 1994.

[63] M. Mühlaüser, editor. *Special Issues in Object-Oriented Programming: Workshop Reader of the 10^{th} European Conf. on Object-Oriented Programming ECOOP'96*. dpunkt, July 1996.

[64] G.C. Necula. Proof-Carrying Code. In *Proc. of the 24^{th} ACM Symp. on Principles of Programming Languages*, Paris, France, January 1997.

[65] M. Nuttall. Survey of systems providing process or object migration. Technical Report Doc 94/10, Dept. of Computing, Imperial College, May 1994.

[66] Object Management Group. *CORBA: Architecture and Specification*, August 1995.

[67] National Bureau of Standards. Data Encryption Standard. NBS FIPS PUB 46, 1977.

[68] J. Ordille. When agents roam, who can you trust? Technical report, Bell Labs, Computing Science Research Center, 1996.

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

[70] J. Ousterhout. Scripting: Higher Level Programming for the 21^{st} Century. Technical report, Sun Microsystems, 1997.

[71] J. Ousterhout, J. Levy, and B. Welch. The Safe-Tcl Security Model. In G. Vigna, editor, *Mobile Agents and Security*, LNCS. Springer, 1998. Also available as Sun Microsystems Technical Report.

[72] A.S. Park and S. Leuker. A Multi-Agent Architecture Supporting Services Access. In Rothermel and Popescu-Zeletin [81], pages 62–73.

[73] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In Rothermel and Popescu-Zeletin [81], pages 50–61.

[74] C. Perkins. IP Mobility Support. IETF draft-ietf-mobileip-16, April 1996.

[75] G.P. Picco, G.-C. Roman, and P.J. McCann. Expressing Code Mobility in Mobile UNITY. In *Proc. of the 6th European Software Engineering Conf.*, volume 1301 of *LNCS*, pages 500–518, Zurich, Switzerland, September 1997. Springer.

[76] M.L. Powell and B.P. Miller. Process migration in DEMOS/MP. In *Proc. of the 6th Symp. on Operating System Principles*, November 1983.

[77] R.L. Rivest. The MD5 Message Digest Algorithm. RFC 1321, April 1992.

[78] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Comm. of the ACM*, 21(2):120–126, February 1978.

[79] D. Rosenblum and A. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of the* 6th *European Software Engineering Conference*, pages pp. 344–360, Zurich, Switzerland, 1997.

[80] D. Rosenblum, A. Wolf, and A. Carzaniga. Critical Considerations and Designs for Internet-Scale, Event-Based Compositional Architectures. In *Proceedings of the 1998 Workshop on Compositional Software Architectures*, Monterey, CA, January 1998.

[81] K. Rothermel and R. Popescu-Zeletin, editors. *Mobile Agents: 1st International Workshop MA '97*, volume 1219 of *LNCS*. Springer, April 1997.

[82] 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*, 1:305–379, October 1988.

[83] T. Sander and C. Tschudin. Protecting Mobile Agents Against Malicious Host. In G. Vigna, editor, *Mobile Agents and Security*, LNCS. Springer, 1998.

[84] B. Schneier. *Applied Cryptography – Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 2nd edition, 1996.

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

[86] B. Smith, S. Yen, and S. Tu. *Introduction to Tcl-DP*.

[87] J.W. Stamos and D.K. Gifford. Implementing Remote Evaluation. *IEEE Trans. on Software Engineering*, 16(7):710–722, July 1990.

[88] J.G. Steiner, B.C. Neuman, and J.I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proc. of the Winter USENIX Conf.*, pages 191–201, February 1988.

[89] M. Straßer, J. Baumann, and F. Hohl. Mole—A Java Based Mobile Agent System. In Mühlaüser [63], pages 327–334.

[90] Sun Microsystems. *Java Remote Method Invocation Specification*, February 1997.

[91] J. Tardo and L. Valente. Mobile agent security and Telescript. In *Proc. of IEEE COMPCON'96*, February 1996.

[92] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. A Survey of Active Network Research. *IEEE Communications*, 35(1):80–86, January 1997.

[93] G. Thiel. Locus operating system, a transparent system. *Computer Communications*, 14(6):336–346, 1991.

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

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

[96] C. Tschudin. OO-Agents and Messengers. In *ECOOP'95 Workshop W10 on Objects and Agents*, August 1995.

[97] G. Vigna. Protecting Mobile Agents through Tracing. In J. Vitek and C. Tschudin, editors, *Proc. of the 3^{rd} Int. Workshop on Mobile Object Systems (MOS'97)*, 1997.

[98] G. Vigna. Cryptographic Traces for Mobile Agents. In G. Vigna, editor, *Mobile Agents and Security*, LNCS. Springer, 1998.

[99] G. Vigna, editor. *Mobile Agents and Security.* LNCS. Springer, 1998.

[100] G. Vigna and L. Bonomi. A Model-Based Electronic Commerce Middleware. In *Proceedings of the International Working Conference on Electronic Commerce*, Hamburg, Germany, 1998.

[101] J. Vitek and C. Tschudin, editors. *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*. Springer, April 1997.

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

[103] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. In Vitek and Tschudin [101]. Also available as Sun Microsystems Laboratories Technical Report TR-94-29.

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

[105] J.E. White. Telescript Technology: The Foundation for the Electronic Marketplace. Technical report, General Magic, Inc., 1994. White Paper.

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

[107] U.G. Wilhelm. Cryptographically Protected Objects. Technical report, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 1997.

[108] 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 [81], pages 86–97.

[109] M. Wooldridge and N.R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2), June 1995.

[110] Y. Yemini. The OSI Network Management Model. *IEEE Communications*, pages 20–29, May 1993.

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

[112] P. Zimmerman. *PGP User's Guide*, March 1993.