Mobility and Remote-Code Execution

Eric Sanchis

University of Toulouse 1,
IUT, 33 avenue du 8 mai, 12000 Rodez, France
sanchis@iut-rodez.fr

Abstract. Using an adapted analysis grid, this paper presents a new reading of the concepts underlying the *mobile code/agent* technology by proposing a decomposition of the paradigms related to *remote-code execution* into three categories: *remote-code calling*, *remote code-loading* and *mobile code*. Models resulting from this decomposition are specified and implemented using a uniform execution system. A distinction between *mobile code* and *mobile software agent* is then proposed.

Keywords: distributed systems, design abstraction, remote-code execution, mobile code, remote code-loading.

1 Introduction

Distributed applications implemented with mobile code or mobile agents were actually developed from the first half of the Nineties [1]. Mainly considered under a technical angle - mechanisms, programming -, opinions are divided today on the utility of mobile agents in these applications [2], [3], [4]. Indeed, as a mechanism of remote-code execution, *mobile code* is considered to be less universal than *remote procedure call* while being more difficult to implement. Moreover, no *killer application* truly emerged. Nevertheless, a certain number of implementations showed that *mobile code* was an interesting mechanism in perfectly targeted applications.

When we wish to study *mobile code* as a manner of executing remote code, we come up against the multiple meanings which are associated with the concept of *code* such as:

- An executable code C provided with its data D and its execution context E: the entity which moves is a complete execution unit defined by the triplet (C, D, E). This type of mobility is generally called *strong mobility*
- An autonomous code or a code fragment C provided with some initialization data D: the couple (C, D) is downloaded on the remote site then executed by a new execution unit (*weak mobility*)
- A non mobile procedure P provided with its parameters which first must be bound to an execution unit before being called (*remote procedure call*).

In these three cases, the concept of *code* has different semantic contents. Moreover, we can notice that neither *weak mobility*, nor *remote procedure call* need the *execution context* to be brought in.

The various interpretations of the notion of *code* and the consideration or not of the *execution context* brought us to revisit the paradigms connected to the distributed execution domain and to re-evaluate the place that the *mobile code* holds in this area. Our work led us to the following conclusion: *weak mobility* and *strong mobility* belong to quite distinct design paradigms. *Weak mobility* just like *remote procedure call* aim at the execution of a remote code, while *strong mobility* is centred on the migration of an execution context. That means that *weak mobility* and *remote procedure call* correspond to the same abstraction that we have called *Remote-Code Execution (RCE)* and which will be detailed in the following sections, whereas *strong mobility* is conceptually close to the migration of process or thread, i.e. a paradigm which we could call *Migration of Execution Unit*. In other words, *weak mobility* is closer to *remote procedure call* than to *strong mobility* because an execution context cannot be reduced to its code part.

In order to position *mobile code* with regard to the architectures generally used within the distributed applications framework, a second reading of the paradigms related to *RCE* was carried out starting from the work and proposals presented in [5].

This paper is structured as follows. Section 2 explains in depth the grid of analysis which was used to carry out our second reading, a grid built on the concepts of *Abstraction*, *Model* and *Mechanism* (called *A2M* grid) where each level of the grid masks a set of non relevant details. Section 3 compares the Fuggetta's design paradigms and the models of our *RCE* abstraction. The last section describes the *execution system* which was used to implement the *RCE* models.

2 Abstraction, Model and Mechanism

The construction of an IT application requires the use of several classes of services provided by the underlying system (communication, synchronization, naming services). In order to extract the principles at the heart of a class of services and to ignore the implementation details, it is preferable to reason about an *abstraction* representing a class of services solving a standard system problem. For example, the conceptual or concrete functioning modalities relative to the services of communication between processes can be grouped together in an *interprocess communication* abstraction. However, it seems that the notion of communication even limited to simple processes takes on very different meanings according to distributed system designers. As far as we are concerned, we define a *communication* as the transmission of a sequence of bytes between a sender process and a local or remote receiver process. Other approaches utilize the concept of *object* or *exchange* (call/reply, transaction) [6], [7], [8]: it is not any more an elementary communication between two processes but a *structured communication* between active entities, with the various additional variations which it authorizes. Consequently, although dealing both with communications, they are two different abstractions. The essential interest of *abstractions* is that they present a global comprehensive view of a problem and their associated solutions (formalized or implemented) when this problem is perfectly limited.

Generally, for each *abstraction* one or more *models* are defined where each one of them formalizes a particular manner to solve the target problem. *A model* is characterized by a set of features, in general a few ones, which identify its specificity and which make it possible to immediately distinguish it from the other models of the same abstraction. Thus, the *interprocess communication* abstraction as we described previously declines according to the two well-known models: the *shared memory* model and the *message-passing* model.

Lastly, while a model shows certain coherence, it can be implemented using very different *mechanisms*. This multiple forms of the same model is due to several factors such as the programming language used or the characteristics of the chosen execution support (resources management, interactions between the execution units, etc.). Thus, inside the Unix family systems, several mechanisms implement the model of *communication by shared memory*. Let us quote for example the communication by *pipe*, *shared memory* or *message queue* (an inadequate denomination with regard to the implemented model!). The properties of these three mechanisms are very different with respect to the communication direction (one-way or bidirectional), to the synchronization of the communicating entities (synchronization provided by the execution support or by the programmer). For complex abstractions such as *RCE*, the associated mechanisms appear under the form of powerful execution systems which section 4 will give some examples of.

It should be noted that often, relations between a model and its implementations are sufficiently strong to introduce a certain *duality* between the model and mechanism concepts. Indeed, certain *mechanisms* studied in a given context, are presented as *models* in another context. For example, the *RPC mechanism* (*Remote Procedure Call*) is often promoted to the rank of model, generally called *Client-Server model*. We can attribute the emergence of this *model/mechanism duality* to the consideration of different elements by the designers to define their paradigms. That will be clarified in section 3 when the study of the paradigms related to the code mobility presented in [5] will be made, paradigms which will be compared with our own models associated to *RCE*.

Moreover, the *plurifunctionality* of certain models/mechanisms used in various contexts leads to a situation where there is no unanimity among the researchers to place a model in a single abstraction. For example, the *Client-Server* model is often considered as a model of communication between process or threads [9], [8], [10] and not as a model of the *Remote-Code Execution* abstraction.

Lastly, some complex mechanisms integrate sub-mechanisms which correspond to models belonging either to the same class, or to different classes. This aspect will be illustrated in section 4 where the *middleware* used to implement the *Mobile Code* model is built on the *Client-Server* model.

In order to dissipate as well as possible the negative effects carried by the *model/mechanism duality* and by the *plurifunctionality* previously described, we proposed an analysis grid *A2M* which is articulated around three adjacent conceptual levels (Figure 1), offering a certain flexibility in the definition and interpretation of the studied paradigms.

The porosity of these three levels already present in the communication between processes is more important when the studied abstraction is more complex such as *RCE*. In order to specify the contents of this abstraction, the following section presents the relationships between the *design paradigms* of a distributed system, *mobile code* and the *RCE* abstraction.

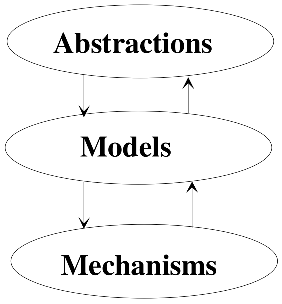

Fig. 1. The A2M grid

3 The Remote-Code Execution Abstraction

A. Fugetta and his colleagues [5] studied in depth paradigms connected to *mobile code*. Their project aimed at clarifying the terms, concepts and technologies which are related to these paradigms. Their work led to the definition of a three-dimension architecture: the *technologies*, the *design paradigms* and the *applications* of mobile code. With regard to these three axes, the second reading we propose focuses on the *design paradigms* (called *models* in the *A2M grid*). This is why only this aspect will be explained in detail thereafter.

3.1 Design Paradigms

These researchers identify four general design paradigms: *Client-Server* (*CS*), *remote evaluation* (*REV*), *code on demand* (*COD*) and *mobile agent (MA)*. In order to clarify the characteristics of these four paradigms, they distinguish the following elements:

- a component *A*, located on site *Sa*, which requests the execution of a service *S* and waits for the corresponding result
- a component *B*, localized on site *Sb*
- the requested service *S*
- the necessary resources *R* (data, files, etc.).

The four paradigms are then decomposed into two categories: *Client-Server* and those which exploit the code mobility (*remote evaluation*, *code on demand* and *mobile agent*).

Using the *Component/Service/Resource* triptych, the *Client Server* paradigm (CS) is stated as follows: at the initial moment, the service *S* and the resources *R* are localized on site *Sb*. The component *A* located on site *Sa* asks the component *B* to execute the service *S*. After execution of S, *B* returns the result to *A*.

The three paradigms associated to the code mobility are described in the following way.

Remote evaluation (*REV*): at the initial moment, component *A* located on site *Sa* possesses the required service *S* but the necessary resources *R* for obtaining the result are present on site *Sb*. Component *A* sends service *S* to component *B* located on *Sb*. *B* uses resources *R* to execute service *S*, then returns the result to *A*.

Code on demand (*COD*): at the initial moment, component *A* has the necessary resources *R* to execute service *S* but this one is located on site *Sb*. *A* interacts with component *B* which sends to it the required service *S*. *A* obtains the expected result by executing on its site the received service *S*.

Mobile agent (MA): at the initial moment, *A* possesses service *S* and a part of the resources *R* necessary to its execution, the other part of the resources being on the site *Sb*. After a partial execution of service *S* on site *Sa*, component *A* provided with service *S* moves on site *Sb* where it continues the execution of *S*.

As it was noticed by the authors, there is an unequivocal division between on one side the *REV* and *COD* paradigms, and on the other side the *MA* paradigm. This separation is attributed to the fact that in the *MA* paradigm, there is not only the movement of a service but the transfer of an execution unit. As we suggested in section 1, this formal asymmetry is due to the consideration of two design paradigms belonging to two different abstractions: the *RCE* and *Execution Unit Migration* abstractions.

Within the framework of a methodology based on the use of the *A2M* grid and before defining conceptually homogeneous design paradigms (models), it is necessary to carefully characterize the contents of the abstraction which will synthesize these models.

3.2 Models of the Remote-Code Execution Abstraction

Defining an *abstraction* consists in clarifying the generic problem, each model of the abstraction providing a specific resolution method. As its naming indicates, the aim of the *Remote-Code Execution* abstraction is the execution of a piece of code present on a remote host. That means that the characterization of the models of this abstraction will be based essentially on the *code* part, and more specifically on the operations on this code such as *copy*, *execution* and *deletion*. To illustrate in a uniform manner the functioning of the various *RCE* models and particularly the flow of control between the local active entity and the remote one, only *synchronous* interactions will be considered. That means that neither asynchronous alternatives, nor various types of *resources* (in all its forms: initialization data, execution contexts - stack, instruction pointer -, opened file descriptors, etc) will not be taken into account in modelling.

A precision must be underlined about the models naming which will follow. As the description of the models is only based on the operations concerning the *code* part, the denomination of the models will be different from the names of the paradigms described in section 3.1. Thus, the *Client-Server* paradigm is named *Remote Code Calling* model in the *RCE* abstraction. This naming has two advantages: on the one hand, it does not use the *Client-Server* expression which is used in many branches of computing and on the other hand it specifically refers to the operation carried out on the *code*, i.e. the aspect which is at the heart of the model. The equivalents of *REV* and *COD* paradigms are considered as two versions of the same principle: the *Remote Code-Loading*. Finally, the *MA* paradigm disappears from *RCE* abstraction for the benefit of a new model: the *physically mobile code* (or more simply *mobile code*).

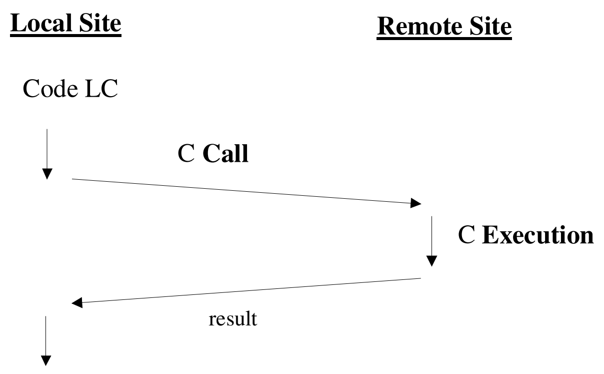

Fig. 2. Remote Code Calling

Remote Code Calling. Figure 2 illustrates the *Remote Code Calling* model.

This model generalizes the traditional *procedure call* mechanism in a distributed system. The principle of *Remote Code Calling* is the following: at the initial moment, the local code *LC* requires the execution of the remote code *C* present on site *Sb* and waits for the result. After execution of code *C* on *Sb*, the result is returned to code *LC* which continues its execution.

It is important to notice that no precision relative to the data, the used resources or the intermediate synchronizations between the execution units associated with codes *LC* and *C* are present into the model: these details are encapsulated into the model implementation according to the used *execution system.*

Remote Code-Loading. It is with the *Remote-Code Evaluation* and *Code on Demand* models that important differences appear with the *REV* and *COD* paradigms.

Remote-Code Evaluation is a generalization of the *evaluation* principle implemented by the interpreters of certain high-level languages. This model can be described as follows: at the initial moment, the local code *LC* copies a code *C* on site *Sb* and asks for its execution there. Code *C* is executed on *Sb*, the result is returned to

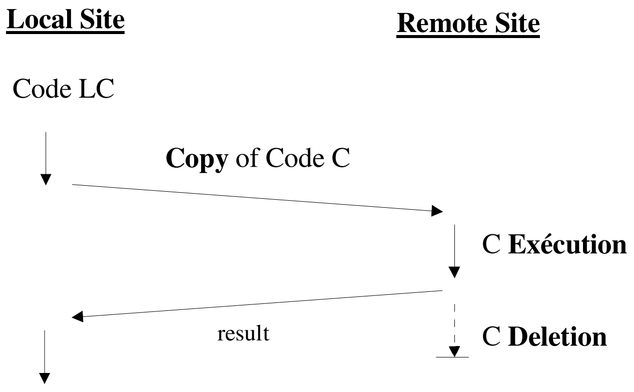

Fig. 3. Remote-Code Evaluation

code *LC* then code *C* is removed from *Sb* (Figure 3). After the result reception, code *LC* continues its execution.

It is the combination of the *Copy/Execution/Deletion* operations which characterizes the remote evaluation of code *C*. This model substitutes the code mobility aspect by a *remote code-loading* sketch. This point of view is perfectly compatible with the application which is usually used to illustrate the *REV* paradigm: the printing of a PostScript file. When a copy of this file is loaded, this copy is directly interpreted by the printer then disappears.

The same operations combination also applies to *Code on Demand*, but executed in an indirect way (Figure 4).

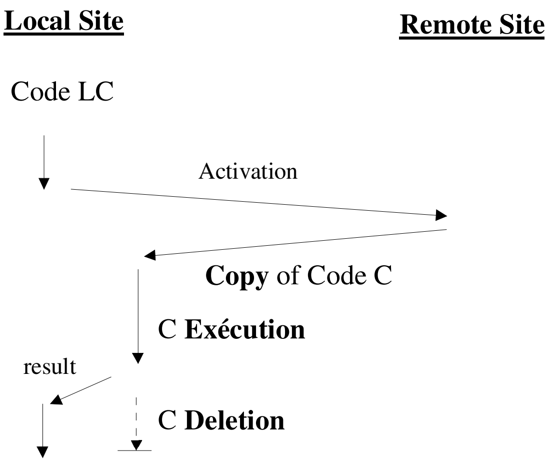

Fig. 4. Code on Demand

The principle of the *Code on Demand* model is the following: at the initial moment, the local code *LC* activates an intermediate remote code *RC* located on *Sb*. Code *RC* copies on *Sa* code *C* which is also located on *Sb*. Code *C* is executed on *Sa*, the result is provided to code *LC* and code *C* is removed from *Sa*. After the reception of the result, code *LC* continues its execution.

The presence of the code deletion in the *Remote-Code Evaluation* and *Code on Demand* models contributes to reinforce the internal coherence of the two models. Indeed, if the operation of deletion was absent, the successive execution of two remote evaluations of the same code *C* on the same remote site *Sb* would lead to a logically incoherent behaviour (copy of code already present).

Mobile code. The *Mobile Code* model derives directly from the *physical mobility* of an object, mobility understood according to the common sense: a mobile object is an object which is present in its physical totality at a point *x* at an instant *t*, then is at a point *y* (*y* different from *x*) at instant *t+1*. This model extends the concept of passive message (*short lifespan data*) to the concept of active message (*persistent lifespan code*).

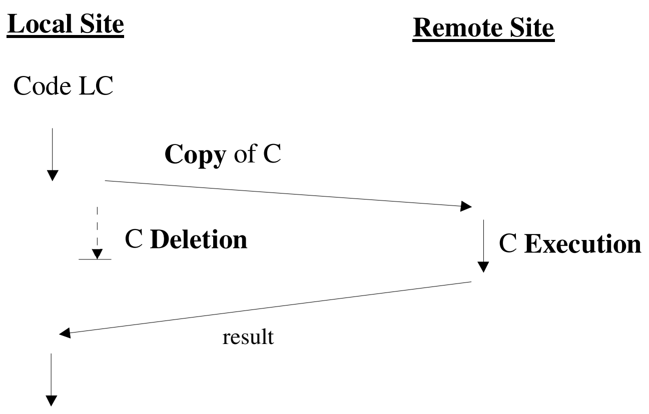

Fig. 5. Mobile Code

The *Mobile Code* principle is translated in the following way: at the initial moment, local code *LC* copies code *C* to site *Sb* then removes code *C* from *Sa*. Code *C* is executed on *Sb* and the result is returned to code *LC* which can continue its execution. Disappeared from site *Sa*, code *C* remains present on the remote location *Sb*: there was an actual physical moving of code between the two sites (Figure 5).

By construction, two successive executions of the same mobile code *C* cannot take place from the same site *Sa*. This behaviour corresponds perfectly to the natural semantics of a code qualified as mobile. The mobility of the code is actual.

Lastly, to be complete and by orthogonality with the *Code on Demand* model, we deduct the indirect version of *Mobile Code* (Figure 6).

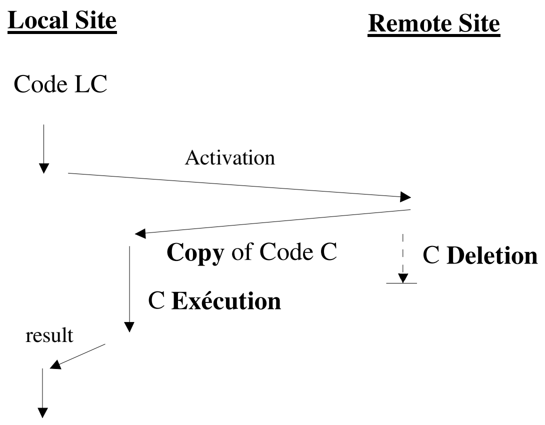

Fig. 6. Indirect Mobile Code

Compared to the *Remote-Code Evaluation* model, in these two models the operations of execution and deletion are made on two different sites.

3.3 Discussion

The *RCE* abstraction as previously characterized, i.e. centred on the code part and the associated operations, has the major consequence to completely reorganize paradigms that are generally related to *mobile code*. Indeed, we showed that:

- *remote evaluation* (paradigm *REV*) and *code on demand* (paradigm *COD*) are two models built on the remotely copying of code and not on its mobility: thus, they must be clearly distinguished from *mobile code*. Our point of view is that neither PostScript printing nor Applet and Servlet technologies constitute examples of *mobile code* but are different implementations of the *remote code-loading* model. On the other hand, advantages generally attributed to *mobile code* or *mobile agent* such as the reduction of the network load or the interest to bring closer code and data to be treated are not related to the mobility but to the copy of the code on the data's site. In other words, certain positive aspects associated to *mobile code* are rather to put at the credit of the *remote code-loading* model
- *strong mobility agents* do not constitute a model of the *RCE* abstraction but a model of a very different abstraction – *Execution Unit Migration* -, an abstraction the study of which (characterization of the abstraction, its models and mechanisms) remains to be made. Consequently, the advantages and disadvantages associated with *strong mobility agents* are not attributable to the management of their code but rather to the mobility of their execution unit as in *process migration* implemented in certain operating systems [11].

The *mobile code* model of the *RCE* abstraction completes in a coherent way the tools offered to the distributed applications designer, tools dedicated to remote-code execution. By construction, this model allows the checking of the uniqueness and the location of the executed code. It also illustrates the difference between *mobile code* and *strong mobility agent*.

Lastly, it invalidates the widespread idea according to which *viruses* and *worms* would be mobile entities. Indeed, the code of these software entities does not move from a site to another but remotely replicates itself on infected sites: if *viruses* and *worms* were mobile, it would be easier to eradicate them. Their power of nuisance results essentially from their capacity of remote replication.

4 From Models to RCE Mechanisms

The purpose of the models presented previously was to extract the specificity of each of them with regard to the other models of the *RCE* abstraction. This specificity was formalized by a particular combination of operations relating to the *code* part, one of the singularities of the suggested interpretation being to consider all other aspects as concerning a particular implementation. Before illustrating a building of each model, i.e. to pass from the model to a mechanism which implements it, the used execution support must first be described. Indeed, it is the characteristics of the underlying *execution system* that will determine the possible synchronizations, the necessary resources and the global modes of functioning.

The judicious choice of the used *execution system* associated with the formal coherence of the *RCE* abstraction models allowed us an implementation of these models which is precise, elegant and homogeneous as well.

4.1 Execution System

Two elements characterize an *execution system*: the *middleware* and the *programming language* which are used.

A *middleware* groups together a set of broad utility services making the implementation of distributed applications easier. Often, a full development system goes with the middleware, a system which not only provides the necessary tools for the construction of the distributed application, but the tools facilitating its deployment too. Two examples of very used middlewares are the RPC and RMI systems. The first one is intended for the implementation of distributed applications written in C language, the second to those written in Java. Both middlewares realize the same remote-code execution model (*Remote Code Calling*), but in different ways: *Remote Procedure Call* for the first and *Remote Method Invocation* for the second.

Two classes of programming languages are mainly used to implement a distributed application: *system programming languages* such as C, C++ or Java and *script languages* such as the Unix shells (bash, sh, ksh), PERL, TCL or Python. Although each class has its advantages and its disadvantages, *script languages* have a strong power of expression perfectly adapted to the use we wish to make [12].

For more simplicity, the *execution system* which was used to implement the various models of the *RCE* abstraction is articulated around the middleware **SSh** and the script language **bash**. The *execution system* **bash/SSh** was selected for the two main following reasons: a native deployment on many computing systems and a concise and elegant programming syntax.

Middleware **SSh** provides a high level network programming interface with the couple of secure commands **ssh/scp**. Well configured, the middleware **SSh** offers a sufficient security level to simply implement the *RCE* abstraction models. The remote-code execution model at the centre of the commands **ssh/scp** is *Remote Code Calling*. By default, interactions via **ssh** or **scp** between the two entities are synchronous. However, the interpreter **bash** natively integrates the necessary mechanisms to implement *asynchronous* interactions.

Shell **bash** - as any other Unix shell - interfaces itself with **ssh/scp** in a perfect manner. Its compactness favours the fast writing of prototypes. Weakly typified and string oriented, it is less sensitive to the traditional problems posed by data representation between heterogeneous systems than system programming languages such as C or other languages of the same family.

4.2 RCE Models Implementation

Supplying an implementation of the *RCE* models simply consists in translating the operations defined in section 3 into comprehensible instructions by the *execution system* **bash/SSh**. These instructions are the following:

To execute the command *cmd* located on the remote site *Sb*: **`ssh Sb cmd`**
To copy on remote site *Sb* the code of command *cmd*: **`scp cmd Sb:`**
To locally copy the code of command *cmd* located on remote site *Sb*:

```
        scp Sb:cmd   .
```

To remove the code of command *cmd*: **`rm cmd`**
To assign to a variable *var* the value resulting from the execution of command *cmd*:

```
        var=$(cmd)
```

The translation of the models is immediate (to simplify, the various cases of error are not treated).

Remote Code Calling.

```
        result=$( ssh  Sb  C )
```

The result coming from the execution of the remote code *C* is assigned to the local variable *result*.

Remote Code-Loading.

Remote-Code Evaluation

```
        scp  C  Sb:
        result=$( ssh  Sb  "C  ;  rm C" )
```

Code *C* is copied to the remote site *Sb*. Then, the two commands *C* and *rm C* are sequentially executed on *Sb*. The result of the remote execution of *C* is assigned to the local variable *result*.

Code on Demand

```
        scp Sb:C  .
        result=$(C)
        rm  C
```

Code *C* located on the remote site *Sb* is locally copied, locally executed then locally removed.

Mobile Code.

```
        scp  C  Sb:
        rm  C
        result=$( ssh Sb C)
```

Code *C* is copied to the remote site *Sb* then the local copy of *C* is removed. Code *C* is executed on site *Sb* and the result is assigned to the local variable *result*.

The indirect version of *Mobile Code* is expressed in the following way:

```
scp Sb:C  .
ssh  Sb  rm C
result=$(C)
```

5 Conclusion

In previous works [13], the study of complex properties such as *autonomy* revealed how important it was to use a precise reasoning framework to be able to analyze then to rigorously classify the models associated with the same paradigm. It was shown that an incompletely controlled abstraction could lead to inappropriate conclusions.

The definition then the use of grid *A2M* applied to *mobility* showed the relevance to distinguish *mobile code* and *strong mobile agent*: the *mobile code* model belongs to the *Remote-Code Execution* abstraction while *strong mobility* is to be associated with the *Execution Unit Migration* abstraction.

The meticulous construction of the *RCE* models clarified the *replication/mobility* duality and underlined the important features making it possible to distinguish the various models of the *RCE* abstraction. The immediate profit was to restore to each model its advantages and its disadvantages.

References

1. Chess, D.M., Harrison, C.G., Kershenbaum, A.: Mobile Agents: Are they a good idea? IBM Research Report, RC 19887 (1994)
2. Lange, D.B., Oshima, M.: Seven Good Reasons for Mobile Agents. Communication of the ACM 42(3), 88–89 (1999)
3. Vigna, G.: Mobile Agents: Ten Reasons For Failure. In: Proceedings of the IEEE International Conference on Mobile Data Management 2004 (MDM 2004), Berkeley, USA, pp. 298–299 (2004)
4. Johansen, D.: Mobile Agents: Right Concept, Wrong Approach. In: Proceedings of the IEEE International Conference on Mobile Data Management 2004 (MDM 2004), Berkeley, USA (2004)
5. Fuggetta, A., Picco, G.P., Vigna, G.: Understanding Code mobility. IEEE Transactions on Software Engineering, 24(5), 352–361 (1998)
6. Goscinski, A.: Distributed Operating Systems – The Logical Design. Addison Wesley, Reading (1991)
7. Silcock, J., Goscinski, A.: Message Passing, Remote Procedure Calls and Distributed Shared Memory as Communication Paradigms for Distributed Systems. Technical Report TR C95/20, School of Computing and Mathematics, Deakin University (1995)
8. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems – Concepts and Design. Addison Wesley/Pearson Education (2005)
9. Tanenbaum, A.: Modern Operating Systems. Prentice Hall, Englewood Cliffs (1992)
10. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts with Java. John Wiley & Sons, Chichester (2007)

11. Thiel, G.: LOCUS operating system, a transparent system. Computer Communication 14(6), 336–346 (1991)
12. Ousterhout, J.K.: Scripting: Higher Level Programming for the 21st Century. IEEE Computer 31(3), 23–30 (1998)
13. Sanchis, E.: Autonomy with Regard to an Attribute. In: IEEE/WIC/ACM International Conference on Intelligent Agent Technology 2007 (IAT 2007), Silicon Valley, USA (2007)