A Characterization of Mobility and State Distribution in Mobile Code Languages

Gianpaolo Cugola[1], Carlo Ghezzi[1], Gian Pietro Picco[2] and Giovanni Vigna[1]

1 E-mail: {cugola | ghezzi | vigna}@elet.polimi.it. Dip. Elettronica e Informazione, Politecnico di Milano, P.za Leonardo da Vinci 32, 20133 Milano, Italy. Tel: +39-2-2399-3638.

2 E-mail: picco@athena.polito.it. Dip. Automatica e Informatica, Politecnico di Torino, C.so Duca degli Abruzzi 23, 10129 Torino, Italy. Tel: +39-11-5647008.

The growing importance of telecommunication networks has stimulated research on a new generation of programming languages. Such languages view the network and its resources as a global environment in which computations take place. In particular, they support the notion of code mobility and state distribution. To understand, discuss, evaluate, and compare such languages, it is necessary to develop an abstract model that allows the meaning of mobility and state distribution to be defined precisely. The purpose of this paper is to provide such a model and to apply it to the analysis of a number of existing new languages.

1 Introduction

Advances in telecommunication networks have given new impetus to research on distributed systems. This research is based on a long term vision where computers are no more viewed as mainly autonomous and self-contained computing devices accessing local resources which, occasionally, communicate with each other; rather, they are part of a global computing platform, built upon a synergy of local and remote resources, whose sharing is enabled by broadband communication networks.

According to this vision, a new generation of distributed applications is being envisioned, whose distinctive feature is the exploitation of some sort of "code mobility". These applications will be called *mobile code applications* (MCAs). By examining current scientific work, the approaches followed to build MCAs can be roughly classified as follows[1]:

- *The "Remote Evaluation" approach.* According to this approach, based on the work described in [77], any component of an MCA can invoke services provided by other components, that are distributed on the nodes of a network, by providing not only the input data needed to perform the service (like in a remote

1. An evaluation and classification of mobile code design paradigms and mobile code applications is the subject of a parallel work, described in [16] and [17].

procedure call scheme) but also providing the code that describes how to perform the service.

- *The "Mobile Agent" approach.* The term "agent" is often abused and rarely defined precisely. In the context of this work, agents can be regarded as *executing units* (EUs). The term executing unit is used hereafter to denote the run-time representation of a single flow of computation, such as a Unix process or a thread in a multi-threaded environment. The adjective "mobile" means that such EUs, running on a given physical network node, can move to a different node where they resume execution seamlessly. MCAs based on this approach are composed of EUs that can move autonomously from node to node in order to accomplish some prescribed tasks.
- *The "Code on Demand" approach.* According to this approach, the code that describes the behavior of a component of an MCA can change over time. A component running on a given node can download and link on-the-fly the code to perform a particular task from a different (remote) component that acts as a "code server".

This paper does not discuss the pros and cons of these new approaches with respect to traditional ones, like the client-server paradigm, since this is out of the scope of this work (see [37] for a preliminary contribution on this issue). Here it will suffice to say that, in principle, these new approaches can provide a more efficient use of the communication resources and a higher degree of service customizing, but raise stronger requirements than traditional ones, for example in the area of security [82][12].

Moreover, the approaches that can be followed to build MCAs demand for dedicated mobile code technology. Traditional mechanisms, like RPC or sockets, are in fact either unsuitable or inefficient for the task. For example, the "Mobile Agent" approach demands for the capability of migrating EUs around a network. This has been investigated by many researchers in the OS [32] and small-scale distributed systems [42] areas, but they are far from being mainstream techniques in large-scale distributed systems.

The approaches described above can be exploited by using the features embodied in a new generation of programming languages, which are usually referred to as *mobile code languages* (MCLs). They can be regarded as languages for distributed systems, whose primary application domain is the creation of MCAs on large-scale distributed systems, like the Internet. MCLs provide facilities and mechanisms for the mobility of EUs and distribution of their state across the network. These languages differ from other languages or middleware for distributed system programming (e.g., CORBA [69] and Emerald [42]) because they explicitly model the concept of separate execution environments and how code and computations move among these environments.

The goal of this paper is to characterize the concepts of mobility and state distribution found in MCLs. To accomplish this, we define an abstract model that is used also as a basis to analyze and compare a number of existing MCLs.

Mobility and state distribution impact heavily on the design of a programming language. Concepts that have been routinely used for traditional programming languages, like scope rules and name resolution, acquire new dimensions in the case of MCLs. In this paper we concentrate only on the essence of mobility and state distribution in MCLs. In a parallel work, we will exploit the result of this work to analyze the impact of mobility on the remaining language features.

In order to provide the reader with a minimum background, the MCLs analyzed are surveyed in Section 2. Section 3 describes in detail our model. Section 4 defines mobility and state distribution concepts and terms. In Section 5 such concepts are used to analyze a set of existing MCLs. Finally, Section 6 describes future directions for the work presented in this paper.

2 A Survey of Mobile Code Languages

This section provides a sketchy overview of the languages that will be discussed in this paper. They are:

- **Java** Developed by Sun Microsystems, Java [33][79] is the language that raised most of the current debate on and expectations from mobile code. It turns out, however, that Java is perhaps the "less mobile" of the languages reviewed here. The original goal of the language designers was to provide a portable, clean, easy-to-learn, and general-purpose object-oriented language. Portability, security and safety features, a programmable mechanism for downloading application code from the network and, most important, a careful marketing that led to integration of a Java interpreter into the Netscape Navigator Web browser, are some of the keys of success of Java as "the Internet language".
- **Telescript** Developed by General Magic, Telescript [91][27] is a rich, object-oriented language conceived for the development of large distributed systems, oriented in particular to the electronic market. Security has been one of the driving factors in the language design, together with the capability of migrating EUs (Telescript threads) while executing. There are two kinds of Telescript EUs: *agents*, i.e., EUs that can migrate to a different site by executing a `go` operation, and *places*, i.e., stationary EUs that can contain other EUs (agents or places).
- **Obliq** Developed at DEC by Luca Cardelli, Obliq [13] is an untyped, object-based, lexically scoped, interpreted language. Obliq objects are local to sites but it is possible to move computations from site to site. Distributed lexical scoping is the glue of such roaming computations, allowing objects distributed on a computer network to be transparently accessed.
- **Agent Tcl** Developed at the University of Darthmouth, Agent Tcl [34] provides a Tcl [71] interpreter extended with support for EU migration. An executing Tcl script can move from one site to another with a single `jump` instruction. A `jump` freezes the program execution context and transmits it to a different interpreter which resumes the script execution from the instruction that follows the `jump`.

- **TACOMA** In TACOMA [41] (Tromsø And COrnell Mobile Agents), the Tcl language is extended to include primitives that allow an EU running a Tcl script to request the execution of another Tcl script on a different site. The script code is sent, together with some initialization data, to the destination site where it is evaluated.
- **M0** Implemented at the University of Geneva, M0 [84] is a stack-based interpreted language that implements the concepts of *messengers*. Messengers, the M0 EUs, are sequences of instructions that are transmitted between hosts and unconditionally executed upon receipt. Each host contains one or more EUs, together with an associative array (called *dictionary*) used to allow memory sharing among different EUs. Hosts are connected by *channels* which represent the communication paths between different hosts.
- **Tycoon** Tycoon [58], developed at the University of Hamburg, is a persistent, polymorphic, higher-order functional language extended with imperative features. All language entities in Tycoon (i.e., values, functions, modules, types, and also threads) have first class status and can be manipulated as standard data.
- **Facile** Developed at ECRC in Munich, Facile [85] is a functional language that extends the Standard ML language with primitives for distribution, concurrency and communication. In [48] a further extension to support mobile code programming is described, which introduces advanced translation techniques and strongly typed resource linking. In the sequel, when talking about Facile, no distinction will be made between the language and its extension.

3 An Abstract Model for Mobile Code Languages

A widely accepted definition for code mobility is still lacking in the research community. The term "mobile code" is often used with a different meaning by different languages (and by different researchers). The same holds for the related concept of state distribution.

The term "mobility" in the context of MCLs intuitively refers to mechanisms to move code, or execution flows (that is, code with state), among different hosts. In the previous sections the term "executing unit" was used informally to describe a running program with an associated state of execution. In this section a more precise characterization of this term is given, together with the set of concepts needed to develop our model. The description will be precise, but informal. A complete formal definition is the subject of our on-going research.

In a conventional sequential programming language, the run-time view of a program is an executing unit (see Section 1) which consists of a *code segment*, that provides the static description of the program's behavior, and a program *state* [31]. The state contains the local data of all active routines together with control information, such as the value of the instruction pointer and the value of return points for all active routines. Control information allows EUs to continue their execution from the current state supporting routine calls and returns.

To provide a conceptual run-time model for mobile languages, the above conventional framework must be extended and modified in the following ways:

1. A concept of *site* must be introduced. A site is a container of *components*. It is an abstraction which is not necessarily mapped onto a host; e.g., two interpreters running on the same host represent two different sites. Components may be *resources* or *executing units*. Resources are passive entities representing data, such as an object in an object-oriented language or a file in a file system. EUs represent the computational elements of our model.
2. The state of an EU can be decomposed into its constituents: the *execution state* and the *data space*. The execution state stores all the control information related to the EU state. The data space comprises all resources accessible from all active routines. For example, a Unix process executing a program `P` written in `C` can be regarded as an EU whose code is the source code of `P` and whose data space is the set of files opened by the process and the set of memory locations the process is able to access, either directly or through a reference, i.e., all the data contained in the stack frames of the routines that were called so far and not yet returned and the heap data reachable through them.
3. It may be useful to identify a subset of the data space, called *data closure*. The data closure of an EU is the set of all local and non local resources that are accessible by the currently executing routine. This data space constituent allows the computation to proceed, possibly calling other routines, but does not support the unwinding of the computation's frame stack upon termination of the current routine.
4. Similarly, it may be useful to identify a portion of the code segment of an EU by defining the *code closure*, which consists of all routines that are directly or indirectly visible from the current one.

Executing units may share part of their data space, that is, two or more EUs may be able to access the same resources. For example, Unix processes may share files, while threads may share memory, too. Moreover, the data space of an EU may include resources located on sites other than the site containing the EU. When this happens, the EU is said to have a *distributed state*.

4 Characterization of Mobility and State Distribution

In conventional languages, like C and Pascal, each EU is bound to a unique site for its whole lifetime. Moreover, the binding between the EU and its code segment is generally static. This is not true for MCLs. Mobile code languages are characterized by the fact that the code segment, execution state, and data space of an EU are able to move from site to site. In principle, each of the constituents identified above can move independently. To support migration, MCLs provide mechanisms to ship code and data towards other EUs and to dynamically link code and data to an existing EU. There are two kinds of dynamic linking: *remote code dynamic linking* and *local resource dynamic linking*.

Remote code dynamic linking naturally extends the notion of deferred linking found in several operating systems to network applications. Remote code dynamic linking allows programmers to implement MCAs based on the "code on demand" approach, that is, applications that download their code dynamically from the network according to different strategies.

Local resource dynamic linking is a mechanism to allow a migrating EU to access resources located on the destination site. Such resources must be linked to their EU's internal representation. Typical examples of resources are represented by files or libraries located on the destination site.

With respect to the form of EU migration supported, two classes of MCLs can be identified: MCLs supporting *strong mobility* and MCLs supporting *weak mobility.*

- **Strong mobility** Strong mobility is the ability of an MCL (called *strong MCL*) to allow EUs to move their code and execution state to a different site. Executing units are suspended, transmitted to the destination site, and resumed there.
- **Weak mobility** Weak mobility is the ability of an MCL (called *weak MCL*) to allow an EU in a site to be bound dynamically to code coming from a different site. There are two main cases for this. Either the EU links dynamically a code segment downloaded from the net or the EU receives its code segment from another EU (that is, the code is explicitly sent from a source site to a destination site). In the latter case, two more options are possible. Either the EU in the destination site is created from scratch to run the incoming code or a pre-existing EU links the incoming code dynamically and executes it.

In both strong and weak MCLs, when the code of an EU is moved, what happens if the names it contains are bound to resources in the source site? In other words, what happens to the whole data space of the source EU (in the case of a strong MCLs) or to the data closure and code closure of the moved code (in the case of weak MCLs) upon migration? Two classes of strategies are possible: *replication strategies* and *sharing strategies*.

- **Replication strategies** can be further divided in:

  Static replication strategy. Some resources, called *ubiquitous resources* [48][56] can be statically replicated in all sites. System variables and user interface libraries are good examples of such resources. The original bindings to such resources are deleted and new default bindings are established with the local instances on the destination site.

  Dynamic replication strategies. A copy of the bound resources is made in the destination site, the original bindings are deleted, and new bindings are established with the copied resources. Two further options exist: (i) remove the bound resources from the source site (*dynamic replication by move*) or (ii) keep them (*dynamic replication by copy*).

- **Sharing strategy** implies that the original binding is kept and therefore inter-site references to remote resources must be generated.

Mobile code languages may exploit different strategies for different resources. Static replication can be used only for stateless resources or for resources whose state has not to be maintained consistent across sites. Dynamic replication by copy is adopted to ensure resource availability both on the source and destination site. Dynamic replication by move is adopted for resources that are neither to be shared nor to be available on both the origin and destination site. Otherwise, when a resource vanishes, a dangling reference may arise. Sharing is adopted for resources that have to be shared among EUs on different sites. The sharing strategy leads to *state distribution*. In fact, when this strategy is adopted, the data space of the migrated EU contains resources located both in the source and destination site.

5 Analysis and Comparison

The languages surveyed in Section 2 differ in the way they support mobility and state distribution. With respect to mobility, TACOMA, M0, Facile, Obliq, and Java are weak MCLs, while Telescript, Tycoon and Agent Tcl are strong MCLs. As for state distribution, only Obliq adopts a sharing strategy and supports distributed data spaces.

In TACOMA, EUs are implemented as Unix processes (the *agents*), while site functionalities are implemented by the Unix operating system with some run-time support. In TACOMA, an agent A1 can require the execution of a new agent `A2` on a remote site. `A1` provides `A2`'s code and initialization data by copying them in a data structure (called *briefcase*) that is sent to the remote site. Upon receipt, a new EU is created with the code provided. The new agent `A2` is able to access the data in the briefcase provided by `A1`, that conceptually becomes part of the receiving site. Since the code sent is not bound to any resource, the problem of data space handling does not arise.

M0 follows the same approach. *Messengers*, (the implementation of the EU abstraction) can *submit* the code of other messengers to remote *platforms* (representing sites). Such code is executed as a new messenger on receipt. The submitting messenger may copy relevant data in the message containing the code submitted, making them available at the destination site.

In Facile, *channels* can be used for synchronous communication between two Facile threads, that are run by different *nodes*, i.e., the Facile run-time support. In this context, threads are EUs and nodes are sites. Channels can be used to communicate any legal value of the Facile language. In particular, functions may be transmitted through channels since they are first-class language elements. The programmer can specify whether the transmitted function is to be directly invoked by the receiver or a new Facile EU is to be spawned using the function code. Since both the data closure and the code closure of the function instance sent may be non-empty, these closures have to be attached to the migrating function. Therefore, dynamic replication by copy is adopted[1]. In addition, static replication is supported for *ubiquitous values* [42]. In Facile, resource dynamic linking is well supported. The programmer can specify interfaces for the resources that a function will access during its lifetime. The function will operate on these resources only through their interfaces. Each interface is composed of a set of function signatures which define the operations that can be performed on the resource. At run-time, any local resource that offers at least a set of operations matching those listed in the interface can be bound to the function. Hence, a function moving to a new site can access any resource on that site among those that match the resource interfaces contained in the function code.

Obliq allows remote execution of procedures by means of *execution engines* which implement the site concept. A thread, the Obliq EU, can request the execution of procedures on a remote execution engine. The code for such procedures is sent to the destination engine and executed there by newly created EUs. Obliq objects are bound to the site in which they are created, and this binding cannot be broken. When an EU asks for the execution of a procedure on a remote site, the references to the objects used by such procedure are automatically translated into network references. Accesses to these objects are translated into callbacks to the originating site. This sharing strategy hides the actual location of the EU data space elements, but the use of network references may results in complex debugging and performance bottlenecks.

Java exploits remote code dynamic linking extensively to enable the implementation of scalable and dynamically configurable applications. The Java compiler translates Java source programs into an intermediate, platform independent, language, called *Java Byte Code*. Java Byte Code is executed by an interpreter that realizes the *Java Virtual Machine* on different hardware and software architectures. The loading and linking of the different classes that compose a Java application are performed at run-time by the *class loader*, which is part of the Java Virtual Machine. Thus, Java provides mechanisms to dynamically load and link part of the code segment of an EU from a remote site that acts as a code repository. If the downloaded code contains references to remote classes their code is automatically loaded when their names have to be resolved for the first time. In terms of the model previously given, this means that the code closure of the downloaded code is dynamically replicated. Since the loaded code is not bound to any resource in the code repository, the problem of data space handling does not arise.

In Telescript, the *engine* embodies the site abstraction. Executing units are *agents* and *places*. Agents can move by using the `go` operation, whose effect is to discard the agent image at the source site and to rebuild it at the destination site[2]. Execution re-

---

1. Facile adopts a sharing strategy only if the communication is established by EUs on the same node, since they can share memory.
2. Telescript provides also a `send` operation that can be used to transmit clones of the sending agent to one or more destination sites.

sumes from the instruction following the go. To determine which part of the data space has to be made available on the destination site, the language formally defines the *ownership* relation between the EUs and the objects they own. During migration, the objects owned by a mobile agent are dynamically replicated by move to the destination site together with the agent code and execution state. The remainder of the data space, composed of the objects referenced by the agent but owned by other EUs, are neither replicated nor shared. In Telescript, resource dynamic linking is achieved through the mechanism of places. Each resource is contained into a place that holds a reference to that resource. When an agent enters a place, it is given a reference to that place, that can be used to invoke methods or manipulate attributes of the place. In particular, the agent may access the resources contained into the place, provided that it has the appropriate access rights.

In Tycoon, EUs are threads. Threads can be moved from a Tycoon virtual machine to another using the migrate primitive. The Tycoon virtual machine embodies the site abstraction. As for the data space of the moving EU, Tycoon adopts dynamic replication by copy. The static replication strategy is also supported through *ubiquitous resources* [56]. In Tycoon, remote resources can be bound to a moving thread dynamically. Migrating threads can specify the type of the remote resources they will access on the destination *migration engine*. When a thread arrives, remote resources of the required type, if present, are bound to the thread and then the thread resumes execution. A special schema is used to allow type checking to be performed at departure-time, in order to prevent exceptions from being raised on the destination site, due to the lack of suitable remote resources. Each thread, in fact, owns a type specification of the destination migration engine that includes the type specifications of the remote resources available. Mismatches between the types of the resources available remotely and the types used in thread scripts to denote such resources are detected at departure-time.

In Agent Tcl, each EU is a Unix process running the language interpreter. The site abstraction is implemented by the operating system extended with the language run-time support that manages the name space of agents and the interactions among different agents. Agent Tcl EUs can either *submit* a new agent to a remote site or migrate to another site. In the first case, the EU provides the code to be executed remotely by a newly created EU. In addition, the programmer can specify explicitly the resources that have to be dynamically replicated by copy on the destination site. In the second case, the migrating EU moves its code, data space, and execution state, except for references to the local file system, i.e., a dynamic replication by move is adopted.

6 Conclusions and Future Work

Mobile code languages are a new trend in programming languages for distributed systems. They can enable brand new applications that can be expected to promote major technological breakthroughs. This work has defined the concepts of mobility and state distribution. Using such concepts, a set of currently available mobile code languages have been analyzed and compared.

We will extend this initial work by refining our model in order to provide a formally defined abstract machine that can be used to specify the semantics of different MCLs.

Acknowledgements

We would like to thank prof. Alfonso Fuggetta for his insightful comments and suggestions about this work.