Analyzing Mobile Code Languages

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

[1] Dip. Elettronica e Informazione, Politecnico di Milano
P.za L. Da Vinci 23, 20100 Milano, Italy
[cugola | ghezzi | vigna]@elet.polimi.it.
[2] Dip. Automatica e Informatica, Politecnico di Torino
C.so Duca degli Abruzzi 24, 10129 Torino, Italy
picco@polito.it.

Abstract 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. To understand, discuss, evaluate, and compare such languages, it is necessary to develop a new set of programming language concepts and/or extend the concepts that are used to deal with conventional languages. The purpose of this paper is to provide such framework. This is done hand-in-hand with a survey of a number of existing new languages.
Keywords. Mobile code languages, distributed programming languages, process migration, run-time model.

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 [16], any component of an MCA can invoke services

[1] An evaluation and classification of mobile code design paradigms and mobile code applications is the subject of a parallel work, described in [5].

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 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, within a *computational environment* (CE), of a single flow of computation, such as a Unix process or a thread in a multi-threaded environment. The adjective "mobile" means that EUs running in a given CE can move to a different CE, where they resume execution seamlessly. MCAs based on this approach are composed of EUs that can move autonomously from CE to CE in order to accomplish some prescribed tasks. The definition of EU is similar to the definition of *mobile object system* given in [1], but our definition is not biased towards the object-oriented paradigm.
- *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 CE 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 [14, 6] 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 customization, but raise stronger requirements than traditional ones, for example in the area of security.

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 [10] and small-scale distributed systems [2] areas, but they are far from being mainstream techniques in large-scale distributed systems.

The approaches described above can be exploited by using the mechanisms 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. These languages differ from other languages or middleware for distributed system programming (e.g., CORBA [13] and Emerald [2]) 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 propose an (initial) set of programming language

concepts, abstractions, and mechanisms, that can be used to analyze, evaluate, and compare existing MCLs or to design new ones. Some are well-known concepts that have been routinely used for traditional programming languages, others are concepts that are shared with traditional languages, and yet become especially relevant or acquire new dimensions in the case of MCLs (e.g., scope rules and name resolution). Still others are typical of MCLs, e.g., the mechanisms to support dynamic linking, mobility, state distribution, and security. In this paper we concentrate on the characteristics that are typical of MCLs. The concepts we will develop are applied in the evaluation of a set of existing MCLs found in literature. In order to provide the reader with a minimum background, such languages are surveyed in Section 2. Section 3 describes in detail our analysis, while Section 4 describes further 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 [11, 22] is perhaps 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 [26, 18] is a rich, object-oriented language conceived for the development of large distributed applications, 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 *engine* by executing a `go` operation, and *places*, i.e., stationary EUs that can contain other EUs (agents or places). Besides the technical issues involved in EU migration, agents and places offer an intriguing and intuitive metaphor for building distributed systems.

Obliq Developed at DEC by Luca Cardelli, Obliq [4] is an untyped, object-based, lexically scoped, interpreted language. Obliq objects are local to interpreters but it is possible to move computations from one interpreter to another. Distributed lexical scoping is the glue of such roaming computations, allowing transparent access to objects distributed on a computer network.

Safe-Tcl Developed by the authors of the Internet MIME standard, Safe-Tcl [3] is an extension of Tcl [21] conceived to support active e-mail. In active e-mail, messages may include code to be executed when the recipient receives or reads the message. Since code is evaluated in the recipient's environment, care must be taken to protect the user from malicious or erroneous code. In the original implementation of the language, mobility was achieved by means of the MIME e-mail run-time support. Safe-Tcl does not provide support for active mail code mobility at the language level and therefore this kind of mobility will not be further analyzed. Instead, a kind of code mobility is achieved indirectly through a programmable dynamic code loading mechanism. Presently, most of the fundamental features of Safe-Tcl have been included in the latest release of Tcl/Tk.

Agent Tcl Developed at the University of Darthmouth, Agent Tcl [12] 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`.

TACOMA In TACOMA [15] (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 host. The script code is sent, together with some initialization data, to the destination host where it is evaluated.

M0 Implemented at the University of Geneva, M0 [24] is a stack-based interpreted language that implements the concepts of *messengers*. Messengers, the M0 EUs, are sequences of instructions that are transmitted between *platforms* and unconditionally executed upon receipt. Each platform contains one or more EUs, together with an associative array (called *dictionary*) used to allow memory sharing among different EUs. Platforms are connected by channels which represent the communication path between different hosts.

Tycoon The overall objective of the Tycoon [20, 19] project, developed at the University of Hamburg, is to offer a persistent programming environment for the development of data-intensive applications in open environments. The Tycoon Language (later on, simply "Tycoon") is used as the application and system programming language in the Tycoon environment. Tycoon 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. Moreover, they are persistently managed by the Tycoon environment transparently to the programmer. Tycoon enables the programmer to use different programming paradigms: functional and imperative programming are supported directly. Moreover, using the higher-order language features, several variants of the object-oriented programming style are supported.

Facile Developed at ECRC in Münich, Facile [7] is a functional language that extends the Standard ML language with primitives for distribution, con-

currency and communication. In [17] 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 Relevant Issues in Mobile Code Languages

Mobility has a strong impact on programming language features. This section analyzes both the characteristics of conventional languages that must be extended to take into account mobility, and the completely new issues that must be considered. For each issue, a general discussion of the problem is presented, followed by an analysis of how such problems have been tackled in the MCLs surveyed in the previous section.

3.1 Type System

The decision about which type system to adopt is one of the most important design choices that have to be kept when defining a new programming language. The language can be defined to be *typeless*, i.e., all data belong to an untyped universe and can be interpreted as values of different types when manipulated by different operators. Typeless languages impose no restrictions on how data are manipulated. The resulting power and flexibility are counterbalanced by the impossibility of protecting data against nonsensical or erroneous manipulations.

At the other extreme, a language may adopt a *strong type system*. In such case, the language definition ensures that the absence of type errors can be guaranteed for a program statically. Languages of this class support the development of verifiable programs. To increase flexibility, however, languages may weaken the type system in such a way that full type checking may only be achieved dynamically, by verifying at run-time that typed data objects are manipulated only through legal operations. This design choice weakens program verifiability, since type errors can only be uncovered if particular input data are used during execution.

In MCLs, strong typing can be practically impossible to achieve for two main reasons. First, code can be downloaded from a remote site and linked dynamically to a running program. Second, resources may be bound to a program as it is executing (see Section 3.3). In both cases, type correctness of a program cannot be verified statically.

MCLs adopt different strategies to weaken their type system in order to allow code and resources dynamic linking.

A first strategy, exploited by several languages, is to complement compile-time type checking with several type checks performed at run-time. Java, Telescript, Facile, and Tycoon are all based on some form of dynamic linking. For instance, Java programs can dynamically download and link code from the network. Type checking is performed at run-time to ensure that this dynamically

downloaded and linked code obeys the language type rules. Similarly, in Telescript, Facile, and Tycoon the run-time system performs type checking to evaluate the type correctness of remote resources accessed when EUs move.

Another strategy, adopted by Obliq and M0, is to perform all type checking at run-time only.

Yet another strategy would be to define a typeless language. Tcl derivatives, like Safe-Tcl, Agent Tcl, and TACOMA are examples of such languages. Tcl variables contain just strings. Tcl instructions elaborate these strings as appropriate. For example, arithmetic operators convert strings to the numeric values they represent, operate on such values and then translate the numeric results back to strings.

3.2 Scoping and Name Resolution

The scope of an identifier is the range of instructions over which the identifier is known. Name resolution rules determine which computational entity is bound to each identifier in any point of a given program.

All the MCLs examined, with the exception of M0, use static scoping rules. This means that the scope of a variable name is determined by the lexical structure of the program. As will be explained later, in M0 the scope of a name can be modified at run time by the programmer.

Name resolution is critical in MCLs. In fact, during the execution of an MCA, the names that appear in the code may be bound to entities that may be located on remote computational environments. Name resolution can either be performed automatically by the run-time support or hooks in the language run-time support can be provided to allow programmers to define their own name resolution rules.

Most of the existing MCLs do not offer particular features to perform automatic name resolution for remote resources. Name resolution for local resources is often performed statically and remote resources have to be accessed explicitly. This means that the programmer is aware of the location of the various remote resources. For example, a Java program can download and link code from the network but the name resolution of dynamically downloaded classes has to be explicitly programmed. A Java programmer must write her own class loader that resolves the names of the classes that have to be downloaded from the network.

In Obliq, new objects created by an EU are stored in the local data space, that is, in the address space of the interpreer where the EU is actually running. The identifier for this object is bound to the storage area of the object. When the EU moves, references to objects in the origin data space are automatically transformed into *network references*. Further accesses to such objects will be transparently translated into callbacks to the interpreter that holds the variable value. Hence, the binding between the object identifier and the object storage area is established at run-time and cannot be changed. Moreover, name resolution rules hide the actual physical location of remote resources to the programmer.

Telescript has a sophisticated name resolution scheme too. As mentioned in Section 2, the migration unit in Telescript is the thread. Each thread, among its attributes, has a list (called *package*) that contains the code for some of the classes it uses. Moreover, the language formally defines the *ownership* relation between the threads and the objects they owns. When an object `Obj` requests an operation that forces the engine to resolve the name of a class (e.g., a `new` operation), the class name is searched into the package of the thread that owns `Obj`. If the class name is not found there, it is searched in the package of the place containing the thread above. If the class name is still missing, it is searched backwards in the enclosing places, until the root of the place hierarchy is reached. Hence, in principle, the name space of a Telescript engine can be enriched by sending "installer" agents that are able to put packages in the appropriate place at run-time, in a fully programmable way.

In M0, the name space of a messenger is defined by its *dictionary stack*. Each dictionary that composes the dictionary stack is an associative array that contains names and their definitions. Whenever an interpreter has to resolve a name in the messenger code, it looks up the dictionaries in the dictionary stack starting from the topmost, hence the first matching definition is used. The dictionary stack can be manipulated by the programmer. In addition to this run-time mechanism, M0 provides, at the language level, a mechanism to force the binding of procedure names in a particular name space. In fact, using the `bind` command, a messenger is able to substitute all operators and procedures names present in a procedure with their corresponding values, according to the messenger dictionary stack. This means that in M0 the scope of a variable can be explicitly programmed.

3.3 Dynamic Linking

Two kinds of dynamic linking are relevant for MCLs: *remote code dynamic linking* and *local resource dynamic linking*.

Code dynamic linking is not a new technique. For instance, many operating systems support dynamically linked libraries (DLLs). The idea underlying DLLs is that libraries do not need to be linked to the main program at compilation time, rather they can be linked at run-time, when they are accessed for the first time. Remote code dynamic linking in mobile code systems naturally extends the notion of deferred linking found in the above 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.

The second form of dynamic linking is *local resource dynamic linking*. When an EU moves from a computational environment to another, it must be able to access resources located on the destination environment. Therefore, resources must be linked to their EU's internal representation. Typical examples of such resources are files or libraries.

Java exploits remote code dynamic linking extensively to enable the implementation of scalable and dynamically configurable applications. As we will

explain in Section 3.6, 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. Classes are loaded and linked only when required. The default class loader loads the required classes from the local host file system, but Java programmers can write their own class loader, implementing different policies. For example, taking advantage of the platform independence of the Java Byte Code, a network class loader that loads classes from a heterogeneous network can be easily implemented.

A similar behavior can be also implemented in TACOMA and Agent Tcl. When a Tcl interpreter does not find a command `c`, it invokes the command `unknown`. This command resembles the Java class loader. By default, it searches for `c` in local packages and libraries, and, if successful, loads and evaluates `c`. As in Java, the `unknown` command can be redefined in order to download a particular package from the network.

In Safe-Tcl the `unknown` command has been removed because it was considered to be unsafe. A different, secure, customizable primitive to dynamically load and link libraries is provided by the language.

The MCLs that exploit resource dynamic linking most are Facile and Tycoon.

In Facile, 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 computational environment can access any resource on that environment among those that match the resource interfaces contained in the function code.

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 engine, 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 Telescript, code dynamic linking is achieved by means of the package mechanism described in Section 3.2. 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.

3.4 Mobility and State Distribution

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.

Since our goal is to analyze and compare different MCLs with respect to mobility and state distribution, it is necessary to abstract away from the details of the various MCLs examined. It is necessary to identify a small set of abstractions that can be used to model the essential aspects of the various solutions provided by different languages.

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 to informally describe a running program with an associated state of execution. In the following section a more precise characterization of this term is given, together with the set of concepts needed to develop our model.

An Abstract Model for Mobile Code Languages This section provides a definition of the abstractions that can be used to provide a run-time model for mobile languages. 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* [9]. 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 *computational environment* (CE) must be introduced. A CE 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 CEs. 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 and may be modeled as the composition of a code segment and a state, as explained above.
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 $\texttt{P}_X$ executing a program `X` written in C can be regarded as an EU whose code segment is the source code of `X`, whose execution state is the program counter and stack structure associated to $\texttt{P}_X$, and whose data space is the set of files opened by $\texttt{P}_X$ and the set of memory locations $\texttt{P}_X$ is able to access, either directly or through a reference. In other words, the data space for $\texttt{P}_X$ contains all the data contained in the stack frames of the routines that were called so far and not yet returned, together with 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 nonlocal 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 CEs other than the one containing the EU. When this happens, the EU is said to have a *distributed state*. Issues related to state distribution will be tackled further on.

Analysis, Classification, and Comparison In this section the MCLs presented in Section 2 are analyzed with respect to the model discussed so far.

In conventional languages, like C and Pascal, each EU is bound to a unique CE for its whole lifetime. Moreover, the binding between the EU and its code segment is generally static. Even in environments that support dynamic linking, the linked code is a resource of the current CE. 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 CE to CE. In principle, each of the constituents identified above (code segment, execution state, and data space) can move independently. Hereafter, we discuss the choices made by the languages surveyed in Section 2. 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 CE. Executing units are suspended, transmitted to the destination CE, and resumed there.

Weak mobility Weak mobility is the ability of an MCL (called *weak MCL*) to allow an EU in a CE to be bound dynamically to code coming from a different CE. There are two main cases for this. Either the EU links dynamically a

code segment downloaded from the net (see Section 3.3) or the EU receives its code segment from another EU (that is, the code is explicitly sent from a source CE to a destination CE). In the latter case, two more options are possible. Either the EU in the destination CE 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 CE? 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* [17, 19] can be statically replicated in all CEs. 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 CE.

Dynamic replication strategies. A copy of the bound resources is made in the destination CE, 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 CE (*dynamic replication by move*) or (ii) keep them (*dynamic replication by copy*).

Sharing strategy implies that the original binding is kept and therefore inter-CE 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 CEs. Dynamic replication by copy is adopted to ensure resource *availability* both on the source and destination CE. Dynamic replication by move is adopted for resources that are neither to be shared nor to be available on both the origin and destination CE. Otherwise, when a resource vanishes, a dangling reference can arise. In addition, dynamic replication is typically adopted for simple values like integers or strings. Sharing is adopted for resources that have to be *shared* among EUs on different CEs. The sharing strategy leads to *state distribution*. In fact, when this strategy is adopted, the data space of the remote EU contains resources located in the source CE.

The languages surveyed in Section 2 differ in the way they support mobility and state distribution. With respect to mobility, TACOMA, M0, Facile, Obliq, Safe-Tcl, 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 CE 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 CE. `A1` provides `A2`'s code and initialization data by copying them in a data structure (called *briefcase*) that is sent to the remote CE. 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 CE. 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 CEs). 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 CE.

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 CEs. 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 Facile is statically scoped, 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[2]. In addition, static replication is supported for *ubiquitous values* [17].

Obliq allows remote execution of procedures by means of *execution engines* which implement the CE 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 CE in which they are created, and this binding cannot be broken. When an EU asks for the execution of a procedure on a remote CE, 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 CE. 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.

As mentioned in Section 3.3 Java provides mechanisms to dynamically load and link part of the code segment of an EU from a remote CE 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

[2] Facile adopts a sharing strategy only if the communication is established by threads on the same node, since they can share memory.

handling does not arise.

As in Java, Safe-Tcl can load routine code dynamically from the network. As opposed to Java, this operation has to be explicitly performed both for the required routine and for the code of the routines called by this one.

In Telescript, the *engine* embodies the CE 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 CE and to rebuild it at the destination CE[3]. Execution resumes from the instruction following the `go`. The ownership concept is used to determine which part of the data space has to be made available on the destination CE. During migration, the objects owned by a mobile agent are dynamically replicated by move to the destination CE 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 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 CE 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* [19].

In Agent Tcl, each EU is a Unix process running the language interpreter. The CE 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 CE or migrate to another CE. 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 CE. 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 dynamically replication by move strategy is adopted).

3.5 Security

The computational environment of an MCL provides a general, distributed platform on which MCAs belonging to different users can be executed concurrently. CEs may host EUs that belong to different users and have different access rights to the hosting environment. In addition, the sites that compose the infrastructure may be managed by different authorities (e.g., a university or a company) and may communicate across untrusted communication infrastructures.

This scenario suggests two security domains: *interCE security* and *intraCE security*.

InterCE security encompasses mutual authentication between a moving EU and the destination CE, as well as integrity and privacy of the communication between two communicating CEs. Migrating EUs need to be authenticated by

[3] Telescript provides also a `send` operation that can be used to transmit clones of the sending agent to one or more destination CEs.

the destination CE in order to determine the identity of the sender. Moreover, moving EU should be able to authenticate the destination CE in order to be protected from spoofing of the destination site [8]. Once that EU and CEs have been mutually authenticated, the EU must be shipped from the source CE to the destination CE. Code travelling among CEs should be protected from tampering and unauthorized disclosure. Integrity mechanisms ensure that the EU representation transmitted over the lines is not modified, either by malicious intent or by errors in the transmission process. Privacy mechanisms ensure that third parties not involved in interCE communication cannot read the information transmitted over the network.

IntraCE security, as noted in [25], encompasses security among different EUs, between EUs and the hosting CE, and between the CE and the supporting operating system. Most of these issues are addressed by access control techniques. Access control of EU to CE's resources is based on the identity of EUs, as determined by the authentication process. Each EU is given a set of access rights to local resources (including other EUs) that is determined by some policy. Typical policies ensure that EUs belonging to different users do not interfere and restrict access to the private data of other EUs. In addition, the ability given to EUs to dynamically link code coming from uncertain and possible malicious source requires strategies that allows *safe execution*, i.e., execution of code in an untrusted environment that protect from abuse of resources accessible under normal circumstances, e.g. sensitive files or EUs belonging to the same user. Safe execution can be ensured by controlling how incoming EUs access local resources and by limiting the number of operations that such EUs are allowed to perform.

One of the most difficult problem in intraCE security is represented by protection of EUs from the hosting CE. Since CEs must execute EUs, they must access their code and run-time representation. It is therefore very difficult, if not impossible, to protect EUs from malicious CEs. Attacks may include denial-of-service, service overcharging, private information disclosure, code and data modification. Usually, in MCLs CEs are considered trusted entities and the problem is not tackled.

As for the security mechanisms, they may be made available to the programmer or not. In the former case, security policies may be programmed. This choice results in greater flexibility, but may increase the language complexity, since concepts like object ownership or thread capabilities must be managed by the programmer. In the latter case, security policies are hardwired in the language run-time support.

Of all reviewed MCLs, Telescript provides the most powerful mechanisms to support security [23]. 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.

In Safe-Tcl, security is based on a *twin interpreter* scheme which consists of a *trusted* interpreter, which is a full-fledged Tcl interpreter, and an *untrusted interpreter*, whose capabilities have been severely restricted, so that one can execute code of uncertain origin without being damaged. The owner of the interpreter may decide to export procedures which are ensured to be safe from the trusted interpreter to the untrusted one. For example, the trusted interpreter could provide procedures to access just a limited portion of the file system. When an e-mail message containing Safe-Tcl code is received, such code is passed to the untrusted interpreter for execution. The untrusted interpreter may require the loading of a Safe-Tcl library. As mentioned in Section 3.4, the loading mechanism can be programmed by the owner of the interpreter. The dynamically loaded code is evaluated by the trusted interpreter and therefore great care must be taken in order to avoid loading of code of unknown or uncertain source.

Java provides a somewhat similar scheme. Java Byte Code received from the network is not directly interpreted by the Java Virtual Machine. It is first scanned by a run-time module, the *Byte Code Verifier*, to check for the absence of potentially dangerous constructs and then it is controlled by a programmable *Security Manager*. Hence, security mechanisms are partly hard-wired into the language run-time support (i.e., the Byte Code Verifier) and partly programmable through the Security Manager.

The run-time support of Agent Tcl and TACOMA provides features to accept mobile EUs from selected hosts only. It provides no mechanisms to manage different EU owners and delegates access rights management and accounting to the underlying operating system. This solution seems to be inadequate for a complex mobile code application.

In M0, integrity of messengers transmitted over channels is supported via checksums included in the messenger message. While no mechanisms are provided for authentication, M0 provides the programmer with privacy and 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.

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.

Facile lacks explicit security mechanisms, but research is on-going on the issue.

3.6 Translation Strategies

The choice of executing a programming language either through direct interpretation or through compilation is not distinctive of mobile code languages. Moreover, this has to do more with the language support environment than with the semantics of the language. Nonetheless, MCAs pose additional constraints and requirements that affect the criteria used traditionally to determine the choice. In fact, code exchanged in an MCA should be:

Portable The target platform is a computer network–be it a LAN or the Internet–and is to be thought of as a set of heterogeneous machines. The obvious goal is to write the mobile code once, and then be able to unleash it for execution on any machine of the target network, without being aware of the hardware and software requirements of each target machine. An interpreted approach can easily support the execution on such an heterogeneous environment, given that an interpreter is provided for each platform involved. 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. If the destination platform is not known, the native code for each platform can be sent to the receiver.

Secure It is necessary to ensure that the code being executed on the target system does not damage system resources, as discussed in Section 3.5. Interpretation allows load-time or run-time checks on the source code in order to verify that only legal instructions are executed.

Mobility introduces additional options for compilation/interpretation. In fact, the code can be interpreted, or it can be compiled either on the sender machine or on the receiver machine. The resources that are actually linked to the code may be different when the code is compiled on one computational environment or on another because of different configurations (e.g., statically linked libraries).

A common strategy among MCLs is the adoption of a hybrid approach, consisting of compiling source programs into an *intermediate language* which is used for transmission and interpretation 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.

Facile supports all the patterns of compilation and interpretation described above. The source code can be transmitted as it is, translated into various intermediate representation with different abstraction levels, or the native code

for the destination platform can be transmitted directly. In addition, to shorten compilation time on receipt in interactive applications, functions are compiled only when the application tries to invoke them, according to a lazy strategy [17].

Java uses a stack-based intermediate language, the *Java Byte Code*, which is interpreted by the *Java Virtual Machine*. The Byte Code is tested by a Byte Code verifier in order to watch for illegal instructions before interpretation. In addition, the Java Byte Code can also be compiled into native code on the fly (i.e., at run-time, when the code is run for the first time) through the *Just In Time* (JIT) compiler[4].

Low Telescript is the name of the stack-based language which constitutes the intermediate language for Telescript (more properly called High Telescript). In opposition to Java, the only purpose of this intermediate form is a more efficient transfer and execution. No security controls are performed, since security in Telescript is ensured at the language level. No compilation of Byte Code is allowed; Low Telescript is strictly interpreted.

Tacoma, M0, Agent Tcl, Safe-Tcl and Obliq are purely interpreted languages.

4 Conclusions and Further 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 analyzed a set of currently available mobile code languages, by proposing an initial set of concepts that can be used to assess and compare different languages and new features that might be added in new designs.

This initial work will be extended in three directions: first, we will extend our work to cover other languages that were not covered here. An example of such languages is Emerald [2]. Second, we will extend and refine our model to provide a formally defined abstract machine that can be used to specify the operational semantics of different MCLs. Work in this area is on-going. Third, we wish to understand what are the main design paradigms that can be followed to design MCAs and how different language features support or enforce such paradigms.

References

1. J. Baumann, C. Tschudin, and J. Vitek, editors. *Proceedings of the* 2^{nd} *ECOOP Workshop on Mobile Object Systems*. Dpunkt, 1996.
2. A. Black, N. Hutchinson, E. Jul, and H. Levy. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1), February 1988.
3. N.S. Borenstein. EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail. Technical report, First Virtual Holdings, Inc, 1994.
4. L. Cardelli. Obliq: A language with distributed scope. Technical report, Digital Equipment Corporation, Systems Research Center, May 1995.

[4] Different implementation of the Java Virtual Machine that perform JIT compilation are provided by several vendors and a JIT compiler was announced by Sun itself.

5. A. Carzaniga, A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Mobile Code Systems through Classification. Technical report, Politecnico di Milano, April 1996.
6. A. Carzaniga, G. P. Picco, and G. Vigna. Designing Distributed Applications using Mobile Code Paradigms. Technical report, Politecnico di Milano, August 1996. Submitted for publication.
7. B. Thomsen et al. Facile Antigua Release Programming Guide. Technical Report ECRC-93-20, European Computer-Industry Research Centre, Munich, Germany, December 1993.
8. D. Chess et al. Itinerant Agents for Mobile Computing. Technical report, IBM Research Division - T.J. Watson Research Center, 1995.
9. C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley and Sons, second edition, 1989. Third ed. forthcoming.
10. A. Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley, 1991.
11. J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, October 1995.
12. R.S. Gray. Agent Tcl: A Transportable Agent System. In *Proceedings of the CIKM'95 Workshop on Intelligent Information Agents*, 1995.
13. Object Management Group. Corba: Architecture and specification, August 1995.
14. C.G. Harrison, D.M. Chess, and A. Kershenbaum. Mobile Agents: Are They a Good Idea? Technical report, IBM Research Division - T.J. Watson Research Center, March 1995.
15. D. Johansen, R. van Renesse, and F.B. Schneider. An Introduction to the TACOMA Distributed System - Version 1.0. Technical Report 95-23, "University of Tromsø and Cornell University", June 1995.
16. J.W. Stamos and D.K. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.
17. F.C. Knabe. Language Support for Mobile Agents. Technical Report ECRC-95-36, European Computer-Industry Research Centre, Münich, Germany, December 1995.
18. General Magic. *Telescript Language Reference*. General Magic, October 1995.
19. B. Mathiske, F. Matthes, and J. W. Schmidt. On Migrating Threads. Technical report, Fachbereich Informatik Universität Hamburg, 1994.
20. F. Matthes, S. Müssig, and J. W. Schmidt. Persistent Polymorphic Programming in Tycoon: An Introduction. Technical report, Fachbereich Informatik Universität Hamburg, 1993.
21. J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
22. Sun Microsystems. *The Java Language Specification*, October 1995.
23. J. Tardo and L. Valente. Mobile Agents Security and Telescript. General Magic Technical Report, 1995.
24. C. F. Tschudin. *An Introduction to the* `MO` *Messenger Language*. University of Geneva, Switzerland, 1994.
25. Jan Vitek. Secure object spaces. In *Proceedings of the* 2^{nd} *ECOOP Workshop on Mobile Object Systems*, July 1996.
26. J.E. White. Mobile Agents. General Magic, 1995.