Mobile Code in .NET: A Porting Experience

Márcio Delamaro[1] and Gian Pietro Picco[2]

[1] Fundação Eurípedes Soares da Rocha
Av Hygino Muzzi Filho, 529, 17525901, Marilia—SP, Brazil
Phone: +55(14) 421-0833, E-mail: `delamaro@fundanet.br`
[2] Dipartimento di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo da Vinci, 32, I-20133 Milano, Italy
Phone: +39-02-23993519, E-mail: `picco@elet.polimi.it`

Abstract. Mobile code systems typically rely on the Java language, since it provides many of the necessary building blocks.
Nevertheless, Microsoft recently released the .NET platform, which includes at its core a virtual machine supporting multi-language programming, and a new language called C#. The competition between .NET and Java is evident, and so are the analogies between these two technologies. From the point of view of code mobility, a natural question to ask is then whether .NET supports mobile code, and how the mechanisms provided compare with those available in Java. This paper aims at providing a preliminary set of answers to this simple question.
The work we report about was not driven by the goal of providing a thorough comparison. Instead, it was driven by the practical need to port an existing toolkit for code mobility written in Java, μCODE, to the .NET environment. This approach forced us to verify our mobile code design on a concrete example, rather than just think about the problem in abstract. The resulting software artifact constitutes, to the best of our knowledge, the first implementation of a mobile code system written for .NET. In the paper, we provide an overview of the .NET mechanisms supporting mobile code, show how they are exploited in our port, and discuss similarities and differences with the Java platform.

1 Introduction

Code mobility [2] is increasingly being considered as part of the mainstream techniques for developing distributed systems. In some cases, code migration takes place behind the scenes, e.g., in middleware like RMI and Jini [4], where code mobility is exploited to increase the flexibility of service invocation. In other cases, the ability to trigger code migration is directly under the control of the programmer. This is the case of a number of systems supporting mobile code, where code can be explicitly relocated from one host to another. In particular, mobile code techniques and mechanisms are typically exploited by mobile agent systems [9], which allow the migration of an entire execution unit (e.g., a thread or a process) to a different host.

The popularity and pervasiveness of mobile code can be ascribed largely to the success of the Java language. Modern technologies supporting code mobility all rely on this language. Even without considering social factors, like marketing and hype, there are a number of technical reasons that justify this phenomenon. In Java, a number of fundamental building blocks for code mobility, like multithreading and communication primitives, are readily available either at the language level or in the standard library. Furthermore, portability and the availability of a programmable class loader opened up unprecented levels of flexibility.

To counter the success of Java as a language for Internet programming, Microsoft has recently released the first version of its .NET [7] platform, in an evident effort to re-gain control of the distributed application market. The .NET environment includes as its core a virtual machine supporting multi-language programming, and a new language called C#.

Natural questions to ask are then what features in .NET can be exploited to support mobile code, and how they compare to those available in Java. This paper aims at providing a preliminary set of answers to this question. The work we report about was not driven by the goal of providing an extensive comparison. Instead, we followed a bottom up approach, driven by the need to port an existing open source toolkit for code mobility written in Java, μCODE [8, 6], to the .NET platform. This approach forced us to verify our mobile code design on a concrete example, rather than just think about the problem in abstract. The resulting software artifact constitutes, to our knowledge, the first implementation of a mobile code system written for .NET.

The paper is structured as follows. Section 2 provides the reader with a concise overview of the .NET platform, and of the features of C# that are relevant to the content of this paper. Section 3 analyzes more specifically those features of .NET that facilitate and support the development of systems involving code mobility. Section 4 provides a minimal description of μCODE, the mobile code toolkit whose port under .NET is described in Section 5. Section 6 elaborates on the findings of our porting experience, and provides some preliminary comparison between Java and .NET as platform supporting mobile code. Finally, Section 7 ends the paper by describing opportunities for future work and by providing some concluding remarks.

2 .NET

.NET is a software platform that aims at providing a complete solution for the development of distributed applications.

The core of .NET is represented by the .NET Framework[1], that provides the base infrastructure for the rest of the platform and unifies the corresponding programming model across several languages.

At the heart of the .NET Framework is the *Common Language Runtime* (CLR) [11], the virtual machine providing the core services of memory management, thread management, compilation, and communication, and handling code

[1] Usually, when people refer to ".NET" they mean the ".NET Framework" rather than the whole platform. Hereafter, we adopt the same convention, since the rest of the paper is concerned only with the .NET Framework.

management and execution in a secure way. The CLR provides language interoperability by defining an intermediate language (called MSIL, Microsoft Intermediate Language) and a single format (called PE, Portable Executable format) for executable code. Compilers generate MSIL code from the specific source language (e.g., Visual Basic or C) and output it into a PE file that contains, besides the MSIL code, metadata that instructs the CLR about how to allocate memory to objects, enforce security, locate referenced modules, and so on. Since PE files are language-independent, code generated from different languages can interoperate easily. The MSIL is similar in concept to Java bytecode, and is a stack-based language with a rich instruction set. While in principle the MSIL could be interpreted, the CLR translates MSIL to native code using a just-in-time compiler. In addition, the CLR also manages the interoperability of managed (MSIL) code and unmanaged (native) code. Unmanaged code is code that does not comply with .NET, and hence does not provide the same guarantees in terms of execution and security. The need for such interoperability typically arises with reuse of legacy code, like native DLL libraries or COM components.

The other fundamental component of the .NET Framework is the *Base Class Library*, a comprehensive set of classes that provide the API towards the services offered by the CLR, together with a lot of other features like graphics, interoperation with DBMSes, collections, I/O, and XML support, to name a few.

The current version of the .NET Software Development Kit includes compilers that produce MSIL from Visual Basic, C++ and C#. The last one is a new language, that constitutes one of the novelty of .NET. C# [10] is an object-oriented language and can be regarded as a high-level version of the MSIL, in that all the features of C# are natively supported by the CLR. On the surface, C# is quite similar to Java. However, it provides some distinctive features[2], of which the most prominent and more relevant to the content of this paper are the notion of delegate, the support for application events as a first-class language construct, and the notion of attribute.

Delegates are similar to interfaces in the sense that they specify a contract between a caller and a specific method. Unlike interfaces, however, delegates are defined at runtime, to create a sort of "instance of a method". Hence, delegates are often used for the same purpose as function pointers in C, e.g., to implement callbacks. For example, in the code below the method `ProcessString` takes as parameter a delegate, whose interface is defined by `StringProcessor`, that is supposed to perform some computation on the string passed as a parameter. In `Main`, the actual method `ComputeLength` is passed as a delegate:

```
public class Example {
  public delegate int StringProcessor(string x);
  public int ComputeLength(string x) { return x.Length; }
  public void ProcessString(StringProcessor x, string y) { x(y); }
  public static void Main() {
    Example e = new Example();
```

[2] More details can be found in the .NET documentation, or in one of the many books on the topic, e.g., [1].

```
    e.ProcessString(new StringProcessor(e.ComputeLength), "abc");
  }
}
```

Delegates are often used in conjunction with events, to specify the handler associated to the occurrence of an event. In C#, the interface of a class can specify which events it can raise, like in the following code fragment:

```
public delegate TempExceededEventHandler(object source, EventArgs e);
public class Sensor {
  public event TempExceededEventHandler TempExceeded;
  ...
}
```

Events can be generated easily, as in `TempExceeded(aSensor,e);` while the delegates describing the behavior of event handlers can be associated to events with statements like

```
TempExceeded += new TempExceededEventHandler(aSensor);
```

It is interesting to note how delegates and events are first-class elements not only in C# but also in the CLR, and hence potentially available to other languages.

Finally, attributes are auxiliary, declarative information that can be associated to given elements of the language, and that get stored in the metadata associated to the compiled code. This information can be later retrieved using reflection, and can be used at runtime. For instance, a `[Serializable]` attribute is used to tag a class field as serializable. A number of predefined attributes are provided, together with mechanisms to create new ones.

3 .NET Features Supporting Mobile Code

The success of Java as a language for mobile code relies on some of its features, which provide the fundamental building blocks for these kind of systems. Roughly, these features can be grouped together as support for concurrency, object serialization, code loading, and reflection.

In this section, we illustrate how .NET supports similar features, highlighting significant departures from Java whenever appropriate. Our description is based on C#, since this is the language we used to develop our port, but most of our considerations should hold also for the other languages supported by .NET.

The content of this section should not be regarded by any means as complete. The API of the .NET Framework provides a huge array of functionality, whose complexity can only be scratched in a short paper like this. Our intent here is to give a concise overview of the fundamental features found in .NET that can be exploited to support the development of mobile code systems and applications. For more technical detail, we redirect the interested reader to the documentation that accompanies the .NET Framework.

3.1 Concurrency

The ability to handle multiple, concurrent activities is fundamental for mobile code systems and especially for those supporting mobile agents. Java-based mobile code systems typically pick threads as their unit of concurrency: for instance, a mobile agent is usually implemented by a thread. A thread is represented in Java as an object of class `Thread`, and executes within the (operating system) process containing the JVM. Java threads are granted shared access to objects residing in the process containing them, while sharing of objects contained in different processes must be handled through interprocess facilities.

At their core, the features provided by .NET to deal with threads are very similar to those found in Java. Threads are represented by objects of class `System.Threading.Thread`, and methods to start, suspend, resume, interrupt, join, abort a thread and get a reference to the current thread are provided, similarly to Java. Interestingly, the thread's code is not bound to reside in a `run()` method, like in Java. Instead, applications can specify at thread creation time which is the method containing the thread behavior by passing a delegate.

The constructs provided for controlling concurrency are instead a little different. The core functionality is provided by the class `Monitor`, that provides features similar to those found in Java's `Object`. The `Enter`, `TryEnter`, and `Exit` methods allow a thread to acquire or release a lock on the `Monitor` object, and thus define a critical region. Moreover, the methods `Wait`, `Pulse`, and `PulseAll` allow a thread to explicitly synchronize with other threads. The `lock` statement provides a syntactic shortcut to define a critical section of code that can be executed only after a lock on an object has been acquired. Hence, `lock` is equivalent to a `synchronized` block in Java. Synchronized methods are instead declared by attaching a `[Synchronized]` attribute to methods.

Several additional utility features are provided, like a `ThreadPool` class for managing collections of threads, a `Timer` class, and several classes (e.g., `Mutex`, `ReaderWriterLock`, and so on) supporting low-level synchronization of concurrent activities.

Nevertheless, the concurrency model put forth by .NET is richer than the Java one since, in addition to processes and threads, it provides the notion of *application domain*, which is a sort of hybrid between the other two. Like threads, application domains are lightweight processes that run in a process. However, unlike threads and similar to processes, application domains cannot directly share code or objects. In essence, application domains are a way to provide isolation between separate applications without incurring the overhead of handling them through multiple processes, hence enhancing performance and scalability. According to the .NET documentation,

> Application domains form an isolation, unloading, and security boundary for managed code.

Hence, not only performance is improved, but the management of application is more flexible, since an application running in an application domain can be stopped and its code unloaded from the system, without having to stop the process containing it. Similarly, different policies for different applications can coexist in the same process. Application domains can be thought of providing a notion of “logical process” inside a “physical” operating system process.

Threads can still be exploited within and across application domains. Nevertheless, since memory cannot be shared across application domains, the programmer is forced to resort to mechanisms similar to interprocess communication. In .NET, these mechanisms are provided by the *Remoting* facility, which provides a form of remote method invocation that can be used not only locally, to cross the application domain boundary, but also to enable communication between applications on remote hosts. Hence, there is a tradeoff between the benefits brought by application domains and the performance overhead and increased complexity when accessing shared resources.

Application domains are available to programmers as `AppDomain` objects. Methods are available to create a new application domain, load code into it, unload an application domain and its code. Moreover, the interface of `AppDomain` also exports two events, `TypeResolve` and `AssemblyResolve`, that can be used to implement code loading schemes, as we describe in Section 5.

3.2 Object Serialization

Serialization is clearly a fundamental building block of mobile code systems based on an object-oriented programming language. It allows to transform a structured object variable into a flat data structure, typically a stream of byte or characters, for subsequent use with an I/O channel, e.g., a socket.

Serializable classes are tagged as such by using the `[Serializable]` attribute, like in:

```
[Serializable] public class Person {
  public String name = "John";
  public String surname = "Doe";
  public int age = 20;
}
```

Notably, this attribute is not inherited. For instance, if a subclass of `Person` is meant to be serializable, the `[Serializable]` attribute must be explicitly attached to it. This is a significant departure from Java, where the a serializable object is declared by implementing the tagging interface `java.io.Serializable`, and hence serializability is automatically inherited by subclasses. Class fields that are not meant to be serialized can be tagged with the `[NonSerialized]` attribute, analogous to Java’s `transient` fields. .NET provides a `ISerializable` interface as well, but with a different meaning from its Java counterpart. In fact, this interface is provided to allow an object to govern its own serialization and deserialization, which is achieved in Java by defining the `writeObject` and `readObject` methods of a class implementing `Serializable`. As in Java, the object code is not stored with the object state. Instead, information about the type of the object is stored with the serialized data, so that the correct type can be retrieved upon deserialization.

In Java, (de)serialization is achieved by using a specific I/O stream class, like `java.io.ObjectInputStream`. Instead, (de)serialization in .NET is delegated to a *formatter* object, that must implement the interface `IFormatter`. For instance, the following snippet serializes an object of type `Person` and writes it in a file:

```
IFormatter f = new BinaryFormatter();
Stream fs = new FileStream("person.dat", FileMode.Open,
                            FileAccess.Read, FileShare.Read);
Person obj = new Person("John", "Doe", 20);
f.Serialize(fs, obj);
s.Close();
```

Two formatter implementations are provided by .NET, providing binary serialization and XML serialization. The latter allows to serialize an object's public properties and fields into an XML file, and is meant to be used for generating human readable descriptions of an object, and for interacting with Web services based on SOAP. Instead, binary serialization is closer to Java serialization, in that it preserves the type of the object, and is typically exploited within the Remoting API for passing parameters in a remote method invocation, similarly to Java RMI. Interestingly, serialization is even more important in .NET, due to the aforementioned impossibility of sharing object directly across application domains. While serialization is exploited in Java only across processes, typically residing across different hosts, in .NET it becomes relevant even in the scope of a single process, to implement object sharing.

3.3 Code Loading

The fundamental mechanism enabling code mobility is the ability to load code dynamically, either into a running application or in a newly spawned concurrent executing unit.

In Java, the unit of code loading is an object type. The bytecode corresponding to a class or interface can be loaded dynamically by the runtime, and more specifically by the class loader, typically when the name of a class that has not yet been loaded is encountered during the execution of an application.

In .NET the unit of code loading is more coarse grained than a single type, and is constituted by an *assembly*. According to the .NET documentation,

> "[Assemblies] form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. [...] To the runtime, a type does not exist outside the context of an assembly."

At first sight, an assembly vaguely resembles a Java JAR file. For instance, each assembly contains a manifest, containing the assembly metadata. Nevertheless, while JAR files are only relevant for deployment, to package together code and resources, assemblies are first class entities in .NET. Assemblies are made accessible to the programmer through the class `Assembly` that, among other things, enables introspection as we discuss later. More importantly, assemblies play a fundamental role in code loading, since they provide the context for the code representing a type.

Assembly loading can be triggered when the runtime attempts to resolve a *reference* to another assembly. References can be defined statically or dynamically. Static references are typically generated by the compiler and stored in the assembly manifest. A typical example is a method call on an object whose class belongs to another assembly. Dynamic references are instead generated when a program requests the CLR to load an assembly that was not referenced statically, or to create and load an assembly from a byte array containing its code. This latter feature is fundamental for implementing mobile code, and can be achieved by invoking either the `Assembly.Load` or the `AppDomain.Load` method. The effect is similar to the `ClassLoader.defineClass` found in Java, where the byte array is reified into a class object.

The sequence of steps performed during assembly resolution is the following:

1. *Determine the assembly version required.* This is performed by consulting a number of system- and application-defined configuration files. Notably, versioning is built in the unit of code loading. This is a significant improvement over Java, where class versioning is still largely an open issue.
2. *Check whether the assembly has been bound before within the runtime.* If yes, the previously loaded assembly is used.
3. *If the assembly is strong-named, check the global assembly cache.* Strong-named assemblies are assemblies that have been signed with a key at creation time. They are meant to be shared among several applications on one machine, and are stored in a machine-wide cache.
4. *Probe for the assembly.* Probing is a process that attempts to determine potential locations for the assembly, by looking at a number of configuration files and applying heuristics.

These sequence of steps cannot be changed directly by the programmer, who can intervene only by manipulating configuration information. In other words, in .NET there is no direct equivalent of the Java programmable class loader. Nevertheless, as we discuss in Section 5, programmable code loading can be provided by a combination of the loading facility provided by the `Assembly` class, the protected name space provided by `AppDomain`, and the reactive features provided by events and delegates.

3.4 Reflection

Reflection, i.e., the ability to obtain type information about an object, is not necessarily exploited within the runtime of a mobile code system, but it is often a fundamental asset for applications that exploit mobile code. Through reflection, an application can instantiate an object using a class that becomes available only dynamically. More importantly, reflection can be used to determine what portion of the class closure must be transferred during code migration.

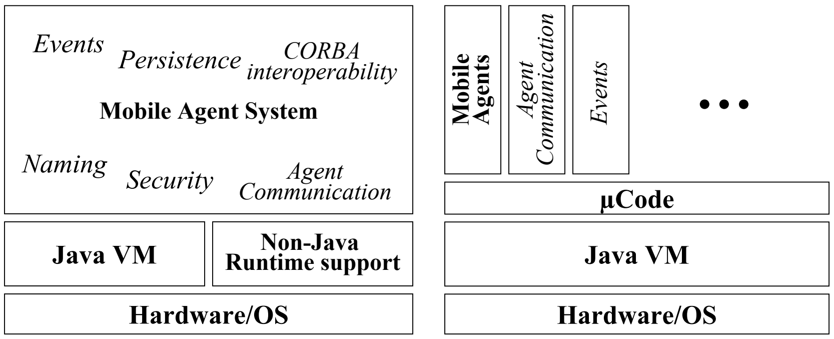

Fig. 1. The architecture of mainstream mobile agent systems (left) and μCODE (right).

At the core of .NET reflection is the `System.Type` class, that provides functionality similar to Java `java.lang.Class`. Thus, for instance, `GetType(String)` returns the `Type` with the given name, similarly to the `Class.forName(String)` found in Java. Once the type is obtained, a new instance of the type can be obtained by retrieving the appropriate constructor (through `GetConstructors`) and by invoking it (through `InvokeMember`). Besides constructors, `InvokeMember` can be used also to invoke methods or get access to members. Through the `Type` class, information can be gathered about the members of a type, its ancestors, the interfaces implemented, and so on. Differently from `Class`, however, `Type` is used not only for classes and interfaces, but also for scalar types, arrays, pointers, and enumerations, since the type system of the CLR is unified.

Nevertheless, as we mentioned before, types are always contained in assemblies. Hence, it is no surprise to find reflective features in the `Assembly` class as well. For instance, `Assembly.CreateInstance(String)` allows to create a new instance of a the named type by invoking the default constructor, after the latter is retrieved from the assembly. Methods that allow to query an assembly for all the `Type`s and resources contained in it are also provided.

In general, the reflection features provided by .NET are more sophisticated (and complex) than the Java counterpart. Part of the motivation lies in the fact that .NET addresses a multilanguage platform, and aims at retaining compatibility with unmanaged code belonging to legacy applications, e.g., COM components. Thus, for instance, a number of methods are also provided that allow to access the internal representation of a type, and a lot of flexibility is provided in defining the rules by which a given member is queried and retrieved.

4 μCode

μCODE [8] is a lightweight and flexible toolkit for code mobility that, in contrast with most of analogous platforms, strives for minimality and places a lot of emphasis on modularity. While mainstream mobile agent systems tend to provide a rich set of features but with a monolithic design, μCODE decouples the core mechanisms supporting code mobility from the other features (see Figure 1). This design achieves modularity, thus improving the understanding and management of the system, and optimizing its deployment.

μCODE revolves around three fundamental concepts: groups, group handlers, and class spaces. *Groups* are the unit of mobility, and provide a container that can be filled with arbitrary classes and objects (including `Thread` objects) and shipped to a destination. Classes and objects need not belong to the same thread. Moreover, the programmer may choose to insert in the group only some of the classes needed at the destination, and let the system downloading and link the missing classes from a remote target specified at group creation time.

The destination of a group is a μServer, an abstraction of the run-time support. In the destination μServer, the mix of classes and objects must be extracted from the group and used in some coherent way, possibly to generate new threads. This is the task of the *group handler*, an object that is instantiated and accessed in the destination μServer, and whose class is specified by the programmer at group creation time. Any object can be a group handler. Programmers can define their own specialized group handlers and, in doing so, effectively define their own mobility primitives.

During group reconstruction, the system needs to locate classes and make them available to the group handler. The classes extracted from the group must be placed into a separate name space, to avoid name clashes with classes reconstructed from other groups. This capability is provided by the *class space*. Classes shipped in the same group are placed together in a private class space, associated with that group. However, if and when needed, these classes can later be "published" in a shared class space associated to a μServer, where they become available to all the threads executing in it, as well as to remote ones.

Class spaces play also a role in the resolution of class names. When a class name C needs to be resolved during the execution of a class originally retrieved from a group g managed by a μServer S, the redefined class loader of μCODE is invoked to search for C's bytecode by performing the following steps: *i)* check whether C is a *ubiquitous class*, i.e. a class available on every μServer (e.g., system classes); *ii)* search for C in the private class space associated with g in S; *iii)* search for C in the shared class space associated with S; *iv)* if t is allowed to perform dynamic download, retrieve C from the remote μServer specified by the user at migration time, and load C; *v)* if C cannot be found, throw a `ClassNotFoundException`.

Moreover, μCODE provides higher-level abstractions built on the core concepts defined thus far. These abstractions include primitives to remotely clone and spawn threads, ship and fetch classes to and from a remote μServer, and a full-fledged implementation of the mobile agent concept.

μCODE is available as open source under the Library GNU Public License (LGPL). Binaries, source, documentation, and examples are available at `mucode.sourceforge.net`.

5 μCode from Java to C#

In this section, we describe the most relevant issues we faced when porting μCODE from the Java platform to .NET. Although the port was developed in C#, most if not all of the content of this section should hold for the other languages supported by .NET.

Clearly, there are different strategies that can be exploited when porting an application to a different platform. In particular, a tradeoff stems from the desire to keep the port as close as possible to the original, and at the same time to use effectively the features of the target environment.

In the work we report here, we strived to implement our port (that we will call henceforth μCODE.NET to distinguish it from the original) so that its API and runtime functionality are as close as possible to μCODE. Nevertheless, we also tried to reach this goal by using the features provided by .NET in the most natural way. This first experience already allowed us to draw some relevant considerations about .NET for mobile code, as discussed in Section 6, and provides the basis for ongoing work in improving our prototype.

As in the original μCODE, the `Group` class allows one to pack objects and code together, and relocate them to a remote μServer. The main difference with the original μCODE, however, is that classes are not shipped individually but as part of an assembly. This is clearly a consequence of the fact that .NET forces types to be always contained in an assembly, and does not provide mechanisms to load the former without the latter. Hence, adding a class to a group automatically triggers also the insertion of the corresponding assembly. A method `AddAssembly`, for explicitly inserting an assembly into a group, is also provided in μCODE.NET.

At the destination μServer, the assemblies in the group are unpacked and made available to the group handler through the private class space. In μCODE, the private class space is realized by associating a customized class loader to the group. The semantics of Java class loading effectively yields separation of name spaces, and hence avoids class name conflicts. In .NET, such isolation is provided by the notion of application domain. For this reason, we decided to perform the unpacking of a group into a newly created application domain. Since applications domains behave like logical processes, the code belonging to different groups is kept isolated.

The new application domain is created by using the methods in the `AppDomain` class. Then, an object is created in this new application domain that is responsible to obtain the group from the μServer, unpack it, register some delegates (as mentioned later), instantiate the group handler, and deliver the group to it. Since application domains enforce code isolation, the instantiation and the access to this "bootstrap" object in the new application domain require the use of the Remoting API.

The disadvantage of this solution is the increased complexity of access to shared resources. Threads created in an application domain associated to a received group can share an object with a thread in another application domain only by using some form of interprocess communication. As a special case, this holds also for access to the public, shared class space associated to the μServer, where code (assemblies) are stored and made available to local and remote nodes. In fact, the μServer runs itself in a separate application domain. In our current implementation, we exploited TCP sockets as a form of interprocess communication. We are currently reworking it to exploit the Remoting API, and simplify object sharing.

Once the application domain is created, assemblies are explicitly loaded from the group into the application domain using `AppDomain.Load`. This method may accept an assembly as a parameter, or even a byte array, from which an assembly is automatically reconstructed. As we already mentioned, this latter feature is similar to the `ClassLoader.defineClass` found in Java. In the Java implementation of μCODE, all the class bytecode coming from other μServers is kept in a hashtable, so that it can be retrieved when the class needs to be transferred again. In fact, a byte array is easily obtained from the file containing the class, but once the bytecode has been reified into a class object there are no means to transform it back in a byte array. For similar reasons, in μCODE.NET we first store the byte array in a hashtable upon arrival, and then load it in the application domain.

The code running in the application domain associated with a group is not necessarily self-contained. Most likely, mobile code is exploited further to enable additional code to be downloaded on demand whenever necessary. In order to load the code associated with the group, the default code loading strategy needs to be redefined. Again, in μCODE this is obtained by redefining the class loader, while in μCODE.NET this ability is provided through application domains. The `AppDomain` class defines two public events that allow the programmer to deal with code loading into the application domain. `TypeResolve` is fired whenever the CLR cannot find the assembly containing the requested type, while `AssemblyResolve` is fired whenever the resolution of an assembly, carried out as discussed in Section 3.3, fails. `AppDomain` defines also the delegate `ResolveEventHandler`, responsible for handling these events. By handling these two events, it is possible to obtain the equivalent of the Java class loader. The following is a snippet of the actual code of μCODE.NET, registering the event handlers:

```
curApp.TypeResolve += new ResolveEventHandler(this.TryToLoadType);
curApp.AssemblyResolve += new ResolveEventHandler(this.TryToLoadAssembly);
```

The methods `TryToLoadType` and `TryToLoadAssembly` are responsible for finding and retrieving the code, and hence for reproducing the original μCODE strategy for class resolution. Below is shown the correspondence between the original strategy used by μCODE and the one implemented by μCODE.NET through the aforementioned delegates:

1. μCODE checks whether the class to be resolved is ubiquitous.
   In μCODE.NET, default ubiquitous assemblies are those of the core .NET runtime, and those containing μCODE.NET itself; other assemblies can be defined as ubiquitous by the programmer. Nevertheless, because of their nature they are assumed to be present in well-known places in the file system. Hence, the delegates need not worry about them, in that these assemblies are

either found in the file system (and hence the aforementioned events would not fire) or, if these assemblies are not, they are not meant to be replaced with foreign code.

2. μCODE searches in the private class space.
   The private class space of a group coincides with its application domain. The assemblies in the private class space have already been loaded in the application domain upon unpacking of the group, so the delegates should never get a chance to get called. Nevertheless, we experienced that, for some undocumented reason, the migrated assemblies unpacked from the group and loaded in the application domain are not found when the application tries to use them for the first time. Hence, when a class or assembly is missing the delegates first search their own application domain in any case.
3. μCODE searches in the shared class space.
   μCODE.NET behaves in the same way, by contacting the μServer that created the application domain and asking for the missing code.
4. μCODE attempts a dynamic download from the address in the group.
   Again, μCODE.NET behaves the same, by looking at the value of the dynamic link source field of the group, and contacting the corresponding μServer.
5. μCODE raises a `ClassNotFoundException`.
   In this case, the delegates return a null value to the runtime, which in turn raises the exception.

Clearly, frequent dynamic class loading may result in communication overhead, or even in the impossibility to proceed with execution, if the code repository is currently unavailable. This is a general issue with mobile code system, and μCODE is one of the very few system that tackle it. In μCODE, the programmer is provided with tools that extend Java reflection with the ability to compute the full closure of a given type. The reflection API of Java allows to capture only the type information associated with the type declaration, i.e., fields, methods, parameters, exceptions, inner classes, superclasses and interfaces. Nevertheless, this constitutes only a fraction of the information required to compute the type closure: types that are used only within the body of a method but are not part of the type declaration are not captured by the reflection primitives. The only way to determine such information is through bytecode inspection, which is performed in μCODE by the `ClassInspector` utility class. This way, the programmer can determine the fraction of the (full) type closure that must be relocated during a code migration, and hence reduce or eliminate the need for dynamic class loading.

In .NET, things are simplified by the role of assemblies as type containers. In fact, by their very nature assemblies already provide a way to pre-package together types that are somehow related. Moreover, the manifest of a given assembly contains information about the other assemblies it depends on, and this information can be easily obtained through reflection from the `Assembly` class, using `GetReferencedAssemblies`. Nevertheless, although information about these dependencies among assemblies gets inserted by the compiler, it must be explicitly supplied by the programmer at compilation or linking time.

6 Discussion

Our presentation thus far has evidenced how .NET provides a number of features that can be exploited for supporting code mobility. At the same time, however, it is also evident how .NET is the result of design criteria that are rather different from those who guided the development of Java.

A significant difference between the two platforms is in the unit of mobility. .NET assemblies define a rather coarse-grained unit, if compared with Java classes. While it is possible to define assemblies containing only one type, it would be cumbersome to do so, since it is the programmer's responsibility to keep track of relationships among assemblies. Moreover, it would be a stretch of the assembly abstraction. Assemblies are really designed to be the unit of application deployment: migrating an assembly is just one of the mechanisms for deploying it. Instead, the ability to relocate a single Java class supports a more flexible (and radical) vision where applications are built out of fine-grained components that can reside in any place of the network, and hence are really distributed. This feature is exploited also in Java-based middleware, like RMI or Jini, to enhance the flexibility of remote method invocation, and allows to pass as parameters objects whose classes are not necessarily pre-deployed at destination. On the other hand, assemblies provide richer metadata information, e.g., including version information, dependencies, and security information, while class versioning is still an largely an open problem in Java.

Code loading, another cornerstone of mobile code systems, is also designed according to significantly different principles in Java and .NET. Java is more liberal about the (re)definition of how code loading is performed. The programmer is free to change the entire class loading behavior, and a class can be loaded by a redefined class loader even when it is accessible from the standard one. Instead, in .NET the programmer has a chance to modify the loading behavior only when the predefined one fails. In other words, while Java supports *proactive* code loading, .NET supports only *reactive* code loading, and in a fashion that always privileges the default loading strategy.

In general, the code loading mechanisms provided by Java are characterized by a lower level of abstraction, if compared to .NET. This results in a high degree of freedom and flexibility when dealing with mobile code, but also in increased complexity: programming with class loaders is often difficult and error prone. In particular, class loaders are only tied to classes, i.e., to the static elements of the language. Instead, with mobile code and especially mobile agents, one would like to associate a different class loading strategy to each executing unit (usually a thread), that is, to the dynamic elements of the language. Nevertheless, achieving this binding in Java is not a straightforward task.

Instead, .NET leverages nicely of the separation among logical processes provided by application domains. Code can be loaded in an application domain, where it becomes part of the code segment of the logical process, and hence separated from the others. This provides a natural and intuitive abstraction for managing mobile code, and is associated to a unit of execution. The disadvantage of this solution is that the code and objects residing in different application domains cannot be shared directly, and instead must be accessed through the Remoting API. This is an issue particularly for mobile agent systems, where agents often seek co-location for the sole purpose of sharing objects. Moreover, the unit of execution chosen may be too coarse grained. Code loading is associated to an application domain, not to a single thread. In our prototype, we chose the most natural solution of associating each group to a separate application domain. However, this may not be the right choice for an application that needs to run in a single application domain and yet leverage of mobile code. Future work will explore whether an alternative design where code loading is realized on a per-thread basis is feasible.

Application domains provide also a nice solution to a key problem in support to code mobility: code unloading. The ability to unload a class is of paramount importance for mobile code, since the codebase evolves dynamically, and yet has a finite size. Unloading becomes critical when the mobile agent paradigm is exploited: the memory of a mobile code server may easily get saturated by a flow of mobile agents, each carrying a different set of classes. Similarly, code unloading is key in dealing with resource-limited devices, like PDAs or cellular phones. In Java,

> A class or interface may be unloaded if and only if its class loader is unreachable. The bootstrap class loader is always reachable; as a result, system classes may never be unloaded. ([5], §2.17.8)

Again, unloading is tied to class loaders, which can be managed in arbitrary ways by the programmer. Moreover, even when the programmer has dealt with class loaders accurately, she cannot have the guarantee that a given class will be effectively unloaded by the JVM. Instead, in .NET unloading is associated to the application domain. When an application domain is discarded, all the assemblies that were loaded in it are unloaded as well. Hence, application domains provide an effective and intuitive mechanism to define the scope of code loading and unloading.

Mobile code systems can be distinguished according to whether they provide support for strong mobility. Strong mobility is defined [2] as the ability of an executing unit to retain the execution state (e.g., the program counter) across migration. Strong mobility is desirable especially for mobile agents, since it makes migration completely transparent to the migrated agents. Nevertheless, it complicates significantly the design of the run-time support, and for this reason it is not supported by Java. Not surprisingly, a similar decision was made also for .NET, although the design of the CLR, and in particularly the ability to store richer metadata information in the bytecode, may be exploited to support strong mobility. Our future activities will explore this line of research.

In any case, the design of the CLR may already simplify the achievement of a desirable feature of mobile code systems: support for multiple languages. Traditionally, this is achieved through ad hoc design of the mobile code platform, like in the case of D'Agents [3]. In .NET, the chore is simplified by the fact that the CLR is designed specifically to accommodate multiple languages. Clearly, the problem of interoperating between different runtimes remains unaltered.

Finally, in this paper we did not consider security issues at all. The .NET platform provides a sophisticated set of security features, that leverage off of the concepts we introduced thus far, like application domains and assemblies. Nevertheless, while we agree that security is a relevant feature in mobile code, our work was driven by the desire to learn first what are the core mechanisms supporting the migration of code, before delving into the details of how to deal with such migration in a secure way.

7 Conclusions and Future Work

In this paper, we reported about an experience in porting an existing mobile code toolkit, called μCODE, from Sun's Java to Microsoft's .NET. The port gave us the opportunity of learning about the features of .NET that can be exploited to support mobile code, and of exploring architectural tradeoffs in implementing mobile code mechanisms in this platform. The software artifact will soon be released publicly, under an open source license.

Future work on the topic of this paper will include the exploration of alternative designs for mobile code, including different code loading schemes, and the provision of strong mobility features in the CLR.

Acknowledgments This work has been partially supported by the "Network-Aware Programming and Interoperability (NAPI)" project, sponsored by Microsoft Research. Márcio Delamaro thanks CNPq (the Brazilian Federal Funding Agency) for supporting his stay at Politecnico di Milano.

References

1. T. Archer. *Inside C#*. Microsoft Press, 2001.
2. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
3. R.S. Gray, G. Cybenko, D. Kotz, R.A. Peterson, and D. Rus. D'Agents: Applications and Performance of a Mobile-Agent System. *Software: Practice and Experience*, 2001. To appear.
4. Jini Web page. `http://www.sun.com/jini`.
5. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
6. μCODE Web page. `http://mucode.sourceforge.net`.
7. .NET Web page. `http://www.microsoft.com/net`.
8. G.P. Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. In *Proc. of Mobile Agents: 2nd Int. Workshop MA'98*, LNCS 1477, pages 160–171. Springer, September 1998.
9. G.P. Picco. Mobile Agents: An Introduction. *J. of Microprocessors and Microsystems*, 25(2):65–74, April 2001.
10. ECMA TC39/TG2. Draft C# Language Specification. Technical report, ECMA, September 2001.
11. ECMA TC39/TG3. The CLI Architecture. Technical report, ECMA, October 2001.