μCode: A Lightweight and Flexible Mobile Code Toolkit

Gian Pietro Picco

Dip. Automatica e Informatica, Politecnico di Torino
C.so Duca degli Abruzzi 24, 10129 Torino, Italy
Phone: +39-11-5647091, Fax: +39-11-5647099
E-mail: picco@polito.it, Web: http://www.polito.it/~picco

Abstract. Although a thorough evaluation of mobile code technology does not exist yet, some studies already evidenced that the powerful (and often heavyweight) abstractions and mechanisms proposed so far are not always flexible enough to fully exploit the benefits of migrating code. μCode is a new mobile code toolkit designed to be flexible, extensible, and lightweight. Its small set of abstractions and mechanisms can be used directly by the programmer or composed in higher-level abstractions—mobile agents included. This paper discusses the fundamental concepts and features of μCode, together with its rationale and motivation.

1 Introduction

Code mobility is becoming increasingly popular among researchers and practitioners due to its expected benefits and to the huge interest revolving around the Java language and the Internet world in general. However, the only pervasive mobile code application is nowadays the execution of *applets* in a Web browser—a rather limited exploitation of the mobile code potential. Most researchers are in fact focusing on the proposal of new technologies, and only very few studies [3, 10] focused on building applications and identifying the benefits and tradeoffs involved. As a consequence, a wider acceptance of code mobility is presently hampered by the fact that the soundness of the abstractions and mechanisms proposed are not verified by quantitative evaluations and experimental evidence.

The goal of this work is to propose a different perspective on the design of language support for mobile code. This is achieved by leveraging off of previous work done by the author in evaluating qualitatively and quantitatively the benefits of mobile code [2, 3], exploiting mobile code in applications [4], and building a conceptual framework for code mobility [6]. The outcome is a proof-of-concept system called μCode, a mobile code toolkit for Java conceived to overcome some of the drawbacks of existing mobile code systems (MCS).

The paper is structured as follows. Section 2 analyzes the rationale and motivations for the development of μCode, discussing the principles underlying its design. Section 3 illustrates the basic concepts of the toolkit, and Section 4 describes in greater detail its features. Finally, Section 5 provides some implementation details and concluding remarks, and discusses options for future work.

2 Rationale and Motivation

μCODE[1] has been designed after research done by the author aimed at identifying precisely the benefits brought by mobile code in the design and implementation of distributed applications. To ground the conclusions in the real world rather than in toy examples, these studies focused on the application domain of network management, adopting both a qualitative [2] and quantitative [3] style of analysis. In particular, the latter defined an analytical model of a management task and derived criteria to compare the effectiveness of client-server and mobile code in reducing the network traffic generated by management. The theoretical findings were verified by measuring the performance of a SNMPv1 [5] implementation against several mobile code alternatives implemented with the Java Aglets API [9]. We chose this platform because it is considered by many researchers as one of the most reliable, supported, and complete. However, the aforementioned studies and the mobile code application described in [4] evidenced the drawbacks discussed in the following[2], and gave rationale to the development of μCODE.

	`C` $\notin$ `CLASSPATH`	`C` $\in$ `CLASSPATH`
`C` $\in$ `AGLET_EXPORT_PATH`	`C` cannot be shipped, but can be fetched	`C` can be shipped and can be fetched
`C` $\notin$ `AGLET_EXPORT_PATH`	`C` can be neither shipped nor fetched	`C` can be shipped but cannot be fetched

Table 1. How to constrain migration of classes with Java Aglets.

Lack of flexibility in the strategies for code relocation. According to the taxonomy for code mobility presented in [6], Java Aglets support weak mobility by enabling shipping and fetching of stand-alone code—the *aglets*. Since the API relies on Java, Aglets support also fetching of code fragments—the aglet classes. However, the strategies that rule class relocation are not completely under the control of the programmer. When an aglet must be migrated, its class must satisfy the constraints illustrated in Table 1. These constraints are specified at setup time by setting the environment variables `CLASSPATH` and `AGLET_EXPORT_PATH`, and cannot be easily changed dynamically. Moreover, there is no means to relocate all the classes needed by an aglet in a single operation—missing classes are retrieved remotely and linked dynamically using the class loader mechanism. An example can clarify why this may be a problem. As discussed by many researchers and confirmed by the aforementioned studies, among the benefits of mobile code and particularly mobile agents is the capability to minimize communication over high-cost links, cost being for instance availability of bandwidth, security concerns, or the money paid to use the link. Mobile agents allow for performing a

[1] μ is the Greek letter for m, initial of *mobile*, thus μCODE is short for *mobile code*. However, μ is also the symbol for *micro*, suggesting that the toolkit is lightweight and small. Along this line, our toolkit can be regarded as the *microcode* for systems providing higher levels of abstraction, much like the microcode in a processor.

[2] Although focused on Java Aglets, most of the remarks apply also to many other Java-based MCSs, like Mole [12] or Odyssey [8].

task remotely in a disconnected fashion, without keeping the client connected during task execution. For instance, on a GSM connection at 9,600 baud it is meaningful to send in one shot all the classes for an aglet to minimize communication on the expensive link and leverage off of disconnected operations. If, in the context of the same application, we are now traversing a local Ethernet, it can be also meaningful to send only the aglet class to minimize the computational overload at loading time on the remote node and exploit the fast link for remote dynamic linking. In other words, *the strategy adopted for the relocation of code is determined by the environment and is not necessarily a property of the code being relocated*. The impossibility to specify dynamically the constraints on relocation of classes and to limit dynamic linking hampers a full exploitation of the mobile agent paradigm.

Level of abstraction. The Java Aglets API is nicely centered around a single, fundamental abstraction: mobile agents. However, an application exploiting mobile code does not necessarily need all the power contained in this abstraction. If this is the case, the semantic richness of the agent concept can backfire on the programmer, who is forced to express lower level concepts in a higher level framework. For instance, imagine a scenario where a site must be upgraded with new classes *proactively* rather than *reactively* as enabled by the Java class loading mechanism. A scheme dispatching an agent on the destination site to install the classes is convenient if several sites must be upgraded at the same time, because the agent can perform the task autonomously. However, this scheme becomes artificial if only a site must be upgraded; using lower level primitives that deal directly with the relocation of classes is more natural. Similar considerations hold for the implementation of a remote evaluation paradigm [11, 6]. In many Java-based systems, the only means to realize it is to create locally and dispatch to destination a mobile agent that has been coded explicitly to handle communication of the remote evaluation results. Again, a different set of primitives may be appropriate, even if they are characterized by a lower level of abstraction.

Communication performance. As evidenced in [3], the size of the code being moved is a key factor in determining the threshold after which code mobility becomes useful in reducing network traffic—another motivation to avoid abstractions enforcing unnecessary verbosity. However, a key factor affecting performance is also the overhead introduced by communication protocols. Network and transport level protocols are usually part of the system requirements, but the language designer retains control on the application level protocol used to transfer mobile code. Existing systems exploit different solutions that range from the direct use of the Java serialization protocol [14] to the use of protocols for distributed objects [8]. Java Aglets employ an Agent Transfer Protocol (ATP) that, as characterized quantitatively in [3], increases significantly the communication overhead under given conditions. Clearly, the overhead of application protocols should be minimized, and in this respect Java serialization provides a good tradeoff between compactness of the encoding and ease of programming and customizability.

Hence, our toolkit is designed around the following principles:

1. *Flexibility.* This has in turn different facets:
   (a) *Control over code relocation.* For the same mobile code, different strategies must be available to cope with different situations.
   (b) *Access to different levels of abstractions.* Although high-level constructs are provided by the toolkit, lower-level constructs must be accessible when some particular or customized functionality is needed.
   (c) *Control over the run-time support.* The run-time support can be controlled directly by programs, and is not an external, opaque program.
2. *Minimality.* The set of concepts and mechanisms must strive for minimality, thus fostering composability and ease of understanding.
3. *Non-invasiveness.* Code mobility is just another option for the programmer, for whom the effort to deal with mobility must be minimized and design freedom maximized. As an example, the programmer should not be forced to subclass from particular classes to use mobile threads. Mobility should be applicable directly to Java threads, thus fostering exploitation of mobile code even in applications that were not initially designed for it.
4. *Extensibility.* μCODE is a thin layer providing core mobile code functionality on top of the Java Virtual Machine (VM). Its design must be open, such that its basic primitives can be composed in higher-level abstractions, or specialized run-time supports can be built.
5. *Portability.* The toolkit must be fully compatible with Java, in order to run seamlessly on all the platforms supported by this language.

Notably, security is missing. Although security is often a major concern in applications exploiting mobile code, this does not hold in general. An example is constituted by network management. The vast majority of organizations still rely on SNMPv1, whose security model is limited to a "community name" transmitted unencrypted along with each SNMP request. In this case, performing local SNMP interactions with mobile code does not provide additional breaches in the management system, because this untrusted code contains the same information exchanged remotely by SNMP, whose rather trivial authentication protocol can be implemented at the application level for mobile code. In this cases the overhead imposed on the run-time support by security is undesirable. For similar reasons, our system does not provide any special abstraction for communication among migrating units. Security and communication mechanisms can be implemented in μCODE either at the application level or by a higher-level mobile code layer, whose development is actually the subject of on-going work.

3 Fundamental Concepts

The basic operations provided by μCODE enable creation and copy of thread objects on a remote μServer, and class relocation among μServers. A μServer is an abstraction of the run-time support and represents a computational environment for mobile threads. Upon migration, thread objects retain their data state and lose their execution state. Thus, only weak mobility is supported, through all the mechanisms identified in the taxonomy [6]; μCODE supports code shipping and fetching of both code fragments and stand-alone code, with both synchronous and asynchronous invocation, as well as deferred and immediate execution of mobile code. As for data space management, no mechanism for resource sharing is in place besides those provided by Java, as this issue is orthogonal to the relocation of code and threads addressed by μCODE. However, a package providing the appropriate mechanisms on top of μCODE is currently under development.

In μCODE, the unit of migration is the *group*. A group is simply a container for classes and objects. This abstraction is reminiscent of TACOMA briefcases [7] or, more closely, of the object-group abstraction found in Sumatra [1]. However, there at least two significant differences:

- In Sumatra, the classes needed by objects in an object-group are not transferred along with the group, and must be shipped explicitly with a separate `downloadClass` operation. The rationale behind this choice is to provide the programmer with finer control on which classes must be provided by the sender and which are already available at destination. This is in accordance with the design principles of μCODE, although it provides a similar operation `shipClass` and its complementary `fetchClass`, but also the capability to insert classes needed at destination directly into the group. The added value coming from this choice is improved atomicity. In the decoupled approach, connectivity may be lost in between class shipping and group transmission. In our approach, either the whole group is reconstructed correctly, or none of its components are available at destination.
- In Sumatra, object-groups cannot contain thread objects. Sumatra designers modified the Java VM to support strong mobility and enable threads to retain their execution state across migration; including threads in an object-group would pose significant semantic problems. μCODE supports only weak mobility to preserve full compatibility with the Java VM. The problems above do not arise because thread objects stored in a group lose their control flow and then become akin to conventional objects.

Classes can be added to a group either individually or collectively by computing the closure of a given class. The *class closure* C^* of a class C is such that:

1. $C \in C^*$;
2. $C' \in C^*$ if $\exists \gamma \in C^*$ such that C' is either: the superclass of γ; the class of a field declared by γ; the class of an exception, parameter, or return value for a method or constructor declared by γ; an inner class declared by γ.

Thus, for example, in the fragment

```
public MyClass extends MySuper implements MyInterface {
  public MyField f;
  public MyReturn myMethod(MyPar p) throws MyException {
    MyVar v = new MyVar();
    class MyInnerClass {...}
  }
}
```

the classes and interfaces `MyClass, MySuper, MyInterface, MyField, MyReturn, MyPar, MyException`, and `MyInnerClass` belong to the class closure of `MyClass`, but `MyVar` does not. If `MyVar` must be transferred as well, it must be added explicitly to the group. Differently from class serialization, managed directly by μCODE, object serialization relies on Java mechanisms and thus requires the class of the object being inserted in a group to implement the appropriate interfaces defined in [14]; an exception is raised otherwise. Java serialization relies on the notion of *object closure*. The object closure of an object o is constituted by o, by all the objects that are fields of o, by all the objects that are fields of the objects above, and so on, recursively. The amount of the closure being serialized can be constrained by declaring a field as `transient`, which prevents serialization of the corresponding object even when this is possible.

The actions taken by a μServer when it receives a group are ruled by two classes associated with the group: the handler and the root. The *handler* class is used to instantiate an object responsible for "unpacking" the group and manipulating its contents. μCODE defines a default handler that implements the basic operations provided by the toolkit. However, programmers can define their own specialized group handlers and, doing so, even define their own relocation primitives. The *root* class provides additional information for handling a group by specifying the class on which the relocation operation must be performed, e.g., to specify which class must be used to spawn a new thread in the destination μServer. The handler and root classes can be any classes and even coincide, and it is not necessary to transmit them along with the group, although in this case it is up to the programmer to ensure that they are found at destination.

After the group has been handled, possibly spawning one or more threads, the corresponding classes are kept into a *private class space*, a name space for classes. The presence of a separate class space avoids name clashes among classes belonging to different mobile threads and guarantees that a mobile thread actually uses the classes shipped with its group rather than some other class with the same name already present on the μServer. However, μCODE provides operations that enable a thread to "publish" classes in the *shared class space* associated with a μServer. There, they become available to every thread hosted by the μServer and can also be accessed by operations shipping or fetching classes performed by threads on remote μServers. Class spaces guide the resolution of class names at loading time. When a class C needs to be resolved during execution of a thread t managed by a μServer S, the μCODE customized version of the Java class loader is invoked and performs the following steps:

1. Check whether C is a *ubiquitous class*, i.e. a class available on every μServer. The set of ubiquitous classes can be changed dynamically. Classes belonging to the Java API and μCODE are ubiquitous by default. Checking ubiquitous classes first prevents redefinition of their behavior.
2. Search for C in the private class space[3] associated with t in S.
3. Search for C in the shared class space associated with S.
4. If t is allowed to perform dynamic download, C is retrieved from the remote site specified by the user at migration time, and loaded.
5. If C cannot be found, throw a `java.lang.ClassNotFoundException`.

[3] Threads created directly by applications and not as a consequence of migration are all associated with the same class space, managed by the default Java class loader.

```
public final class Group {
  public void addClass(String className)
  public void addClassClosure(String className)
  public void addObject(String label, Object obj)
  public Object getObject(String label)
  public void addMobile(Mobile threadObj)
  public Mobile getMobile()
  public int ship(String destination, String dynLink, boolean synch)
}
public interface GroupHandler {
  public abstract Runnable unpack(Group group, MuServer server)
}
public interface Mobile extends Runnable, Serializable {
}
```

Fig. 1. The class `Group` and the interfaces `GroupHandler` and `Mobile`.

4 Programming with μCode

μCode is structured along three layers of abstraction. All the abstractions and mechanisms provided by μCode are built using the implementation of the concepts described in the previous section, i.e., groups and their handling, and class spaces. Thus, the core of μCode are the classes `Group` and `ClassSpace`, and the interfaces `GroupHandler` and `Mobile`, described in the following. This first layer of core functionality, although still available to the programmer, is used to compose operations that handle relocation of classes and threads at a more convenient level of abstraction. These are the operations made available by the `MuServer` class, that also provides access to the underlying run-time support. Finally, a third layer is constituted by user abstractions composed using those provided by the other layers; the mobile agent abstraction is an example. This section gives a brief description of the API provided by μCode.

The Class `Group` and the Interfaces `GroupHandler` and `Mobile` The methods of `Group` are shown[4] in Fig. 1. `addClass` inserts in the `Group` instance the class corresponding to the given `className`, specified with the usual Java convention for packages. Already existing classes are overwritten. `addClassClosure` inserts the whole class closure rooted at `className`. `addObject` inserts a given object in the group. The parameter `label` provides a way to tag objects for later retrieval on the remote μServer, using the complementary method `getObject`. The methods `addMobile` and `getMobile` are a variant of the above that insert and retrieve an object implementing the interface `Mobile`, shown in Fig. 1. `Mobile` extends `Runnable` and `Serializable`, and thus represents an object that can contain a thread of control and yet be transferred on a different μServer. The aforementioned methods are just a more convenient way to manage the common case where a single thread object is moved. In fact, although a group can contain more than one `Mobile` object, only the one inserted with `addMobile` is handled automatically by the system; a customized handler is required for the others. The handler class can be any class implementing the interface `GroupHandler`, shown in Fig. 1. This interface defines a single method `unpack` called automatically by the run-time support when the group is received. This method returns

[4] The declarations of exceptions are omitted for the sake of clarity.

```
public final class ClassSpace {
 public boolean containsClass(String className)
 public Class getClass(String className)
 public void removeClass(String className)
 public void copyClassTo(String className, ClassSpace classSpace)
 public void copyClassFrom(String className, ClassSpace classSpace)
 public void moveClassTo(String className, ClassSpace classSpace)
 public void moveClassFrom(String className, ClassSpace classSpace)
}
```

Fig. 2. The class `ClassSpace`.

an object implementing the interface `Runnable` defined by Java, i.e. an object that can be used to spawn a new thread of execution. However, a `null` object can be returned when group handling does not require a new thread, e.g., to provide new class relocation primitives. Finally, the `ship` method provides relocation of the `Group` on the μServer specified by `destination`, identified by a pair `hostname:port`. The `dynLink` parameter enables dynamic linking of classes from a remote μServer, thus providing direct support for a code on demand paradigm [6]. This way, the group is allowed to provide just a subset of the classes required for its handling and the execution of the associated thread objects, if any. If and when additional classes are needed, they will be downloaded from the location specified by the `dynLink` parameter and linked by the remote class loader. A `null` value for this parameter prevents remote dynamic linking. All the classes that are not transmitted along with the group but are indeed needed for group handling must be present at destination or, in alternative, remote dynamic linking must be enabled. In any other case, an exception `java.lang.ClassNotFoundException` is raised on the destination μServer, as described in the previous section. The `synch` parameter determines whether the `ship` operation is performed synchronously or asynchronously. If `synch` is true, the operation is blocking for the caller, that is suspended until a return value is received after group handling on the remote μServer. If `synch` is false, the method returns immediately after group transmission is completed.

The Class `ClassSpace` The methods of `ClassSpace`, in Fig. 2, enable management of the contents of a class space by testing the presence of a class in a class space, getting the class object of a class belonging to the class space, removing the class from the class space, as well as moving and copying a class across different class spaces. These features are useful when performing group handling, and in general to modify the set of classes that are available either to a given thread or to all the threads sharing a class space in a μServer.

The Class `MuServer` This class provides the abstraction of a μServer, and collects methods that provide code mobility on top of the classes described so far. The first three methods in Fig. 3 support remote cloning and creation of threads, and `rSpawnGroup` is the most general operation. The contents of the `group` parameter transferred on the remote μServer are completely up to the programmer. When the group is received at destination, if no custom handler is specified, the default handler looks into the received group for a thread object by performing a `getMobile`, and spawns a new thread of control for it. If

```
public final class MuServer implements Runnable {
  public static final int NONE
  public static final int ROOT
  public static final int FULL
  public int rSpawnGroup(String destination, Group group,
                         String dynLink, boolean synch)
  public int rSpawnThread(String destination, String className,
                          int classClosure, String dynLink,
                          boolean synch)
  public int rCopyThread(String destination, Mobile threadObj,
                         int classClosure, String dynLink,
                         boolean synch)
  public int fetchClass(String destination, String className,
                        int classClosure, boolean synch)
  public int shipClass(String destination, String className,
                       int classClosure, boolean synch)
  public static MuServer getServer()
  public ClassSpace getPrivateClassSpace()
  public ClassSpace getSharedClassSpace()
  public Group createGroup(String rootClass, String handlerClass)
  public void boot(int port)
  public void shutDown()
}
```

Fig. 3. The class `MuServer`.

the thread object is not found, it is created using the root class. The parameters `destination`, `dynLink`, and `synch` retain the meaning defined for the `ship` method in `Group`. The creation of a `Group` object must be requested to a specific μServer, through a `createGroup` operation; in fact, resolution of the handler and root classes depend on the contents of the class spaces associated to a particular μServer. `rSpawnThread` creates a new thread at the specified μServer and makes it easier to build the corresponding group, by enabling the programmer to specify just the `className` of the object for which a thread is going to be spawned. This class actually constitutes the root class of the group being sent by μCODE. In addition, the class closure rooted at `className` can be included. The parameter `classClosure` can assume the values `NONE` if no class should actually be transmitted, `ROOT` if only the `className` class should be transmitted, and `FULL` if the full class closure should be transmitted. Finally, `rCopyThread` clones the `Mobile` object `threadObj` on a remote μServer, discarding the execution state. By default, the classes being transferred as a consequence of the invocation of the three operations above are cached on the destination μServer in the private class space associated with the resulting thread. These classes can be moved to the shared class space using the methods provided by `ClassSpace`; a reference to these spaces is acquired using the methods `getPrivateClassSpace` and `getSharedClassSpace`. Also, a reference to the `MuServer` object hosting the thread can be acquired with the `getServer` method; the result is `null` for threads that have not been created as a consequence of migration.

The methods `shipClass` and `fetchClass` enable relocation of classes across different μServers. Both operations can access only the shared space of the destination μServer, as required by the semantics of private class spaces. In any case, since no built-in naming scheme to identify threads is present in μCODE, there are no means to fetch or ship a class directly to the private class space of a thread running on a remote μServer. If needed, this kind of mechanism

```
public abstract class MuAgent implements Mobile, GroupHandler {
  private transient MuServer server = MuServer.getServer();
  public MuAgent(MuServer server) {this.server = server;}
  public final int go(String destination, int classClosure,
                      String dynLink, boolean synch) {
    String className = this.getClass().getName();
    Group group = server.createGroup(className, className);
    group.addMobile(this);
    switch (classClosure) {
    case MuServer.NONE:
      break;
    case MuServer.ROOT:
      group.addClass(className);
      break;
    case MuServer.FULL:
      group.addClassClosure(className);
      break;
    }
    server.rSpawnGroup(destination, group, dynLink, synch);
    Thread.currentThread().stop();
  }
  public final Runnable unpack(Group group, MuServer server) {
    MuAgent agent = (MuAgent) group.getMobile();
    agent.server = server;
    return agent;
  }
}
```

Fig. 4. The implementation of the class `MuAgent`.

can be built on top of μCODE. `shipClass` copies one or more classes from the current μServer to the shared class space of the remote μServer. Analogously, `fetchClass` copies one or more classes from the shared class space of the remote μServer to the one of the μServer hosting the thread requesting the operation. In both cases, the class(es) remain available at their source μServer. As before, the amount of code transferred is determined by the `classClosure` parameter, although the value `NONE` now raises a `MuCodeException` because the semantics of the operations prescribe that at least one class must always be transferred.

The method `boot` activates the run-time support listening for incoming mobile code on the specified port; multiple μServers can co-exist on the same machine and even in the same Java VM. Nevertheless, this is needed only if the application needs to receive and execute incoming mobile code. In the case where the application only needs to send mobile code, *no additional run-time support is needed* besides the Java VM and the classes in the μCODE package. Additional methods of `MuServer`, not shown in Fig. 3, allow for specifying ubiquitous classes and setting properties like compression and communication timeout.

The Class `MuAgent` The class `MuAgent` provides a natural abstraction for implementing a mobile agent paradigm with a technology providing weak mobility. Movement is reduced to a single instruction, the `go` method, with parameters similar to the ones in `MuServer` methods. In particular, a `MuAgent` instance can perform dynamic linking from a remote μServer. `MuAgent` exemplifies how the abstractions provided by μCODE can be composed in higher-level abstractions. Figure 4 shows the whole implementation of class `MuAgent`, except for minor details. The interesting part is the `go` method body. It retrieves first the class name of the `MuAgent` subclass for the object the method is actually bound to. The information about the class name is then used to create an instance of `Group`. The method body adds to it the thread object to which the `go` is bound, using `addMobile` as described earlier. When the group is received at destination, the run-time support present at that node calls the implementation of the method `unpack`, shown in Fig. 4, that retrieves the thread object using the complementary method `getMobile` and returns it to the run-time support, that spawns a new thread of execution for it. After adding itself to the group, the method `go` adds the amount of the class closure requested by the user. Then, it uses `rSpawnGroup` to actually send the group on the remote μServer. Finally, it stops the current thread, that will be eventually garbage collected.

5 Discussion and Future Work

This paper presented μCODE, a mobile code toolkit designed and developed entirely by the author to provide flexible, extensible, and lightweight support for code mobility in distributed applications developed using Java. The rationale for μCODE comes from research developed by the author and his colleagues in evaluating the benefits of mobile code in real world application domains. μCODE is being effectively used to implement applications for active networks [4] and network management. The current implementation of μCODE consists of less than a thousand non-commented source statements, which generate approximately 40Kbytes of bytecode. Additional figures are not yet available, but the use on the field of μCODE showed that it provides adequate flexibility and a reduction in communication overhead.

The implementation of μCODE leverages off of some advanced features provided by the Java language. The computation of the class closure employs the reflective features provided by Java [13] to retrieve information about the members of a class. However, the Java reflection mechanism cannot detect the need for classes that are declared in method bodies, as exemplified in Section 3. To work around this limitation would involve on-the-fly inspection of the class bytecode—a major endeavor with respect to the benefits achievable. The object serialization mechanism provided by Java has been redefined to provide a specialized handling of classes during serialization and customized class loading during deserialization. Class loading is fundamental to the provision of class spaces. Each μServer contains at least two class loaders: a default class loader servicing requests for classes present on the system and one servicing requests for the shared class space. When the μServer is handling a group, a new class loader is created for it. Due to the semantics of class loading in Java, every subsequent class loading triggered by the classes resolved through this class loader will go again through it, thus preserving a separate class space.

The notion of class space plays a key role in improving flexibility. Through the shared class space, the set of classes that are visible to other μServers can be redefined dynamically by the applications rather than being hard-wired in some configuration file. Furthermore, the distinction between the shared and private class spaces enables the definition of mechanisms for class "caching" on remote μServers, leaving the control in the programmer's hands rather than to some predefined policy of the system. The idea of class space is reminiscent of the *dictionary* concept found in `MO` [15], in that it provides direct access to the name space of an executing unit. The coincidence is not by chance. Like `MO`, μCODE aims at providing a set of basic abstractions and mechanisms to build higher-level systems. Unlike `MO`, however, these features rely directly and completely on the Java language, rather than on a complicate Postscript-like notation.

On-going work on μCODE is developing extensions providing higher-level abstractions and mechanisms, addressing in particular resource sharing, communication, and security. At the same time, the toolkit is being used in several project and its performance is being characterized quantitatively and compared to that of other systems.

Acknowledgments The author wishes to thank Gianpaolo Cugola and Giovanni Vigna for their precious comments on an early draft of this work.

References

1. A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 111–130. Springer, April 1997.
2. M. Baldi, S. Gai, and G.P. Picco. Exploiting Code Mobility in Decentralized and Flexible Network Management. In *Mobile Agents: 1st International Workshop MA'97*, volume 1219 of *LNCS*, pages 13–26. Springer, April 1997.
3. M. Baldi and G.P. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proc. of the 20th Int. Conf. on Software Engineering*, pages 146–155, April 1998.
4. M. Baldi, G.P. Picco, and F. Risso. Designing a Videoconference System for Active Networks. In this proceedings.
5. J.D. Case et al. Simple Network Management Protocol. RFC 1157, May 1990.
6. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5):342–361, May 1998.
7. D. Johansen, R. van Renesse, and F. B. Schneider. An Introduction to the TACOMA Distributed System—Version 1.0. Technical Report 95-23, Tromsø and Cornell Univ., June 1995.
8. J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4):21–30, 1997.
9. D.B. Lange. Java Aglets Application Programming Interface (J-AAPI). IBM Corp. White Paper, February 1997.
10. M. Ranganathan et al. Network-Aware Mobile Programs. In *Proc. of the USENIX 1997 Annual Technical Conf.*, January 1997.
11. J.W. Stamos and D.K. Gifford. Remote Evaluation. *ACM Trans. on Programming Languages and Systems*, 12(4):537–565, October 1990.
12. M. Straßer, J. Baumann, and F. Hohl. Mole—A Java Based Mobile Agent System. In *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conf. on Object-Oriented Programming ECOOP'96*, pages 327–334. dpunkt, July 1996.
13. Sun Microsystems. *Java Core Reflection*, January 1997.
14. Sun Microsystems. *Java Object Serialization Specification*, February 1997.
15. C. Tschudin. *An Introduction to the* `MO` *Messenger Language.* Univ. of Geneva, Switzerland, 1994.