Supporting Disconnected Operation in a Mobile Object System

Marco Túlio Valente[1,2], Roberto Bigonha[1],
Mariza Bigonha[1] and Antônio Alfredo Loureiro[1]

[1]Department of Computer Science, Federal University of Minas Gerais
[2]Institute of Informatics, Catholic University of Minas Gerais
Belo Horizonte, MG - Brazil
E-mail: {mtov,bigonha,mariza,loureiro}@dcc.ufmg.br

Abstract

With the increasing use of mobile devices to access the Internet anytime and anywhere, the relevance of disconnected operation has emerged. Disconnected operation allows users to execute applications during temporary failures in networks or when they explicitly decide to work off-line. This paper presents a system, called Jamp, that uses mobile computation (or logical mobility) to handle disconnections. Jamp has abstractions supporting the migration of groups of objects and classes to other nodes of the network. In this way, programmers can *push* applications to clients with code and data that makes disconnected operation possible.

1 Introduction

Nowadays, advances in wireless networks and portable hardware technology are making *mobile computing* possible. Users carrying handheld devices, like personal digital assistants (PDAs), can now access the Internet anytime and anywhere. In this new paradigm, disconnected operation is a relevant requirement, i.e., users should be able to continue executing their applications during temporary failures in the network or when they explicitly decide to work off-line [12]. The reason is that wireless networks are subjected to higher error rates and larger bandwidth fluctuations than fixed networks. In addition, voluntary disconnections are often requested by users to reduce communication costs.

Previous works in distributed systems have showed that caching of data is a central idea to disconnected operation [7, 8]. The reason is that cache can be used not only to improve performance but also to enhance availability. However, in mobile computing systems it is important to send to clients not only data but also code. In this way, it is possible to *push* to clients applications that can execute in the absence of the network. For example, an e-business site can pro-actively send applications that allow mobile users to choose off-line the item they want to buy. Similarly, a conference reviewing system can *push* to the members of the program committee code and data to support reviewing of papers in disconnected mode.

This paper describes a system, called Jamp, that uses mobile computation to handle disconnections. Mobile computation is the notion that the execution of a program need not be tied to a single node of the network [4]. In Jamp, mobile computation is used to distribute to clients software that can execute in disconnected mode. Thus, the system uses mobile computation (or logical mobility) to handle the problems raised by the physical mobility of portable computing devices.

When compared to other mobile computation systems, Jamp shows differences related to the mobility and communication model supported by the system:

- Mobility model: In Jamp there is an abstraction, called container, to the construction of applications robust to disconnections. A container is a group of objects and classes that can be pro-actively shipped to execution environments provided by nodes of the fixed network or by mobile devices. In Jamp, these environments are organized into a two level hierarchy in order to deal with disconnections that can constraint the migration of containers and to support operation across several administrative domains.
- Communication Model: Jamp does not make use of any construction that requires continuous connectivity or that creates "static bindings" that can restrict the migration of containers.

The remainder of the paper is organized as follows. Section 2 describes the programming model supported by Jamp. Section 3 describes the system API and shows examples of its use. In Section 4, the implementation of the system is described. Section 5 reviews related work and Section 6 concludes the paper.

2 Programming Model

This section describes the mobility and communication constructions that exist in Jamp to deal with connectivity failures in the Internet.

2.1 Mobility Model

The main construction available in Jamp is called *container*. A container is a group of objects and classes that can be shipped to another node of the network. The system provides methods to create containers and also to insert and remove objects and classes from containers. The option to move objects and classes provides support to operation in disconnected mode since we can transfer in a single operation all data and code that an application needs to execute. Once the transfer is completed, the application no longer depends on the network to run. The programmer, however, can make the choice to move only objects in containers when the destination node already has the code to run the application. In this way, the network load is reduced.

In order to receive containers from other nodes, stationary computers or mobile devices should provide a *context* to this execution, i.e., contexts are processes available in some nodes of the network to receive and execute containers. In Jamp, contexts can also provide *resources* to this execution like, for example, a data structure.

In the wireless Internet, however, it is not realistic to suppose that it is always possible to move a container directly from any context A to another context B. The first reason is that B may be running in a mobile device that is not connected to the network when the migration is requested.

And the second reason is that B can be in an administrative domain different from A and a firewall protects access to B.

In order to solve these problems, the mobility model of Jamp organizes the contexts of the network into a two level hierarchy. The first level is composed by *system contexts* and the second level by *user contexts*. A system context is a service that can receive containers from other contexts and then proceed in one of the following ways: start a new thread for the execution of the container or store the container in secondary memory. The later case is chosen when the container is shipped to a specific user of the system context. In the proposed model, system contexts should run in a node that is always connected to the Internet and it should be possible to send containers to this node from other administrative domains.

The second level of the hierarchy of contexts is composed by *user contexts*. Every user context is associated to a system context and to a user of the application. From time to time, a user context retrieves all containers shipped to its user that are stored in the associated system context. In Jamp, however, the execution of containers in user contexts does not start automatically after downloading them. In the system, the user has to explicitly request this execution using the graphical interface of the user context. Since user contexts do not need to execute continuously neither need a reliable connection to the Internet, they can run in mobile devices.

The mobility model of Jamp can be compared to the one used by electronic mail systems, which is the oldest and most popular *push* application available nowadays in the Internet. By this analogy, a container can be compared to an electronic mail, with the advantage that containers have code and data and not only text. System contexts can be compared to mail servers and user contexts can be compared to mail reader systems.

The mobility model of Jamp can also be compared to the one used by Java applets [1], which is the most used mobile code application in the Internet. By this analogy, a system context is similar to a Web server where applets are stored. A user context can be compared to a Web browser where applets are executed and a container is similar to an applet. But, unlike applets, containers are robust to disconnections, since they provide support to both code and data mobility. Containers can also be pro-actively shipped to users in order to support the construction of *push* applications. The communication model of Jamp, described in the next section, is also more flexible than the *sandbox* model used by applets in Java.

2.2 Communication Model

In Jamp, objects communicate by calling methods, as usual in object oriented languages. A mobile object, i.e., an object that is part of a container, can hold references to objects of the same container and to objects of another container. Since in the Internet it is not possible to suppose continued connectivity, references in Jamp can be in two states: connected and disconnected. The meaning of these states is the following:

- A reference is *connected* when it references an object located in the same context as the object that holds the reference.
- A reference is *disconnected* when it references an object located in a different context from the object that holds the reference.

In Jamp, connected references can be used to call methods of the objects they reference. But when a method is called using a disconnected reference, an exception is raised. Therefore, the

system uses references to name objects, but only connected references can be used to call methods of the named object.

This is the fundamental difference between the communication model of Jamp and the one normally used in distributed object systems, where proxies encapsulate access to remote objects. However, it is only feasible to support transparent access to remote objects in environments with low frequency of disconnections, like local area networks [4]. Moreover, proxies require non-official TCP/IP ports for remote communication and firewalls usually forbid any network traffic through these ports.

Contexts in Jamp can also provide resources to the execution of containers. A resource is a non-mobile object and thus can not be added to any container. Resources have a name, given by the programmer when the resource is created. Similar to references to mobile objects, a reference to a resource can also be connected or disconnected. A reference to a resource with name n is connected when there is a resource with the same name in the current context; otherwise, the reference is disconnected.

Figure 1 gives an example of how connected and disconnected references work. In this figure, a and b are references to mobile objects (represented by squares) and c is a reference to a resource (represented by a diamond). In case 1, there is a container running in context P and the references a, b and c are connected. Case 2 shows the configuration of the system when the container moves to context Q. In this new situation, references a and c are connected but reference b is disconnected. If the container returns to context P, case 1 is reestablished.

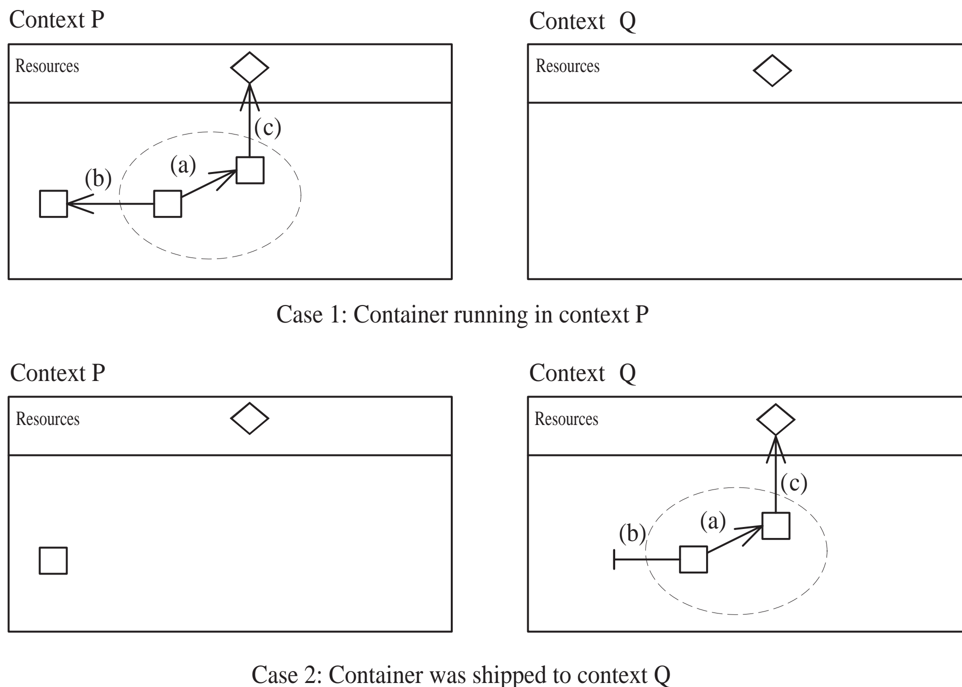

Figure 1: Connected and disconnected references

The communication model of Jamp allows objects in one container to hold references to objects in another container. Object sharing across containers is important to do not constraint the implementation of mobile applications. If object sharing is not supported, cross container communication should use only copy-by-value semantics, which can be inefficient in some cases. Besides, copy-by-value can make complex the distribution of mutable objects, i.e., objects whose state change continuously, like application environment objects [2].

In Jamp, however, object sharing does not create "static bindings" that restrict the migration of containers. The reason is that cross-container references have their state changed after migration. For example, in Figure 1, the state of reference b changed from connected to disconnected after the migration of the designated container.

3 Jamp API

Jamp implements in Java the programming model described in Section 2. The system is structured as a package with the following classes: `JContainer`, `JResource`, `JSystemContext` and `JUserContext`.

3.1 Containers

In Jamp, containers are objects of the `JContainer` class, whose public methods are described in Table 1[1].

Class JContainer
`JContainer(String description)` Creates a new container with a description about it
`void addObj(Object obj)` Adds the specified object to the container
`void addClass(String className)` Adds the code of the specified class to the container
`void removeObj(Object obj)` Removes the specified object from the container
`void removeClass(String className)` Removes the specified class from the container
`void move(String contextName, JStartObject startObj)` Moves the container to the specified context. The execution in the new context will start by the method `run` of the specified object
`static Object newObject(String className)` Creates a new mobile object of the specified class

Table 1: Public methods of class `JContainer`

In Jamp, mobile objects are created by the static method `newObject` of class `JContainer` and not by the `new` operator of Java. Also the class of a mobile object must implement at least one interface. References to mobile objects should be declared using one of these interfaces and not

[1] Exceptions are omitted from the methods for the sake of clarity.

using the class itself. As described in Section 2 such references can be in two states: connected or disconnected. Moreover, in Jamp mobile objects can be in only one container at a time.

The following example shows the creation of a container and the instantiation of two mobile objects of classes `PaperImpl` and `ReviewFormImpl`. Next in the code, these two objects, their classes and a third `ReviewFrameImpl` class are added to the container. This third class can be used, for example, to create an object remotely. Last, the `move` method is used to send the container to another context.

```
JContainer container= new JContainer ("paper01");
Paper p= (Paper) JContainer.newObject("PaperImpl");
ReviewForm r= (ReviewForm) JContainer.newObject("ReviewFormImpl");
container.addObj(p);
container.addObj(r);
container.addClass("PaperImpl");
container.addClass("ReviewFormImpl");
container.addClass("ReviewFrameImpl");
container.move("john@server.foo.br:5000", r);
```

The `move` method provides support to the so-called *objective migration* of containers, i.e., containers in the system can only be moved from the outside of the container [5]. Besides, since the JVM does not support migration of threads, the `move` method only operates in containers that are not active, i.e., no threads should be running in the objects of the container when the migration is requested. Otherwise, an `ActiveContainerException` exception is raised by the `move` method. In the design of the system the alternative solution of stopping all the threads running in the container was not chosen because it could leave its objects in an inconsistent state.

The name of the destination context in the `move` method is specified in the format `user@host:port`, where `user` is the name of the destination user of the container, `host` is the name of the machine where the destination context is running and `port` is the number of the TCP/IP port associated to this context. In order to start a thread for the execution of the container after its arrival instead of storing it in disk, the name of the context in the `move` method should be specified in the format `host:port`, i.e., without a user name.

3.2 Resources

A resource is a non-mobile object that provides services to containers running in a context. Resources have a name and clients should know this name before using the resource. In Jamp, resources are objects that implement the interface `JResource`. This interface only contains a method `getName()` that returns a string with the name of the resource.

A client object can declare fields that are references to resources. These fields should be initialized with an object that stores the binding between the field and the designated resource. This object is created using the static method `newBinding(resourceInterface, resourceName)` of class `JResBinding`. In the following example, the fields `buffer` and `calendar` are references to resources.

```
class A_Impl implements A {
   BufferResource buffer;
   CalendarResource calendar;
```

```
    void init() {
      buffer= (BufferResource) JResBinding.newBinding("BufferResource", "buffer01");
      calendar= (CalendarResource) JResBinding.newBinding("CalendarResource",
                "calendar2001");
    }
  .....
}
```

In this example, the `init` method calls the `newBinding` method to initialize the resource fields of the class with information about the resources accessed by them.

3.3 Contexts

In Jamp, contexts are created using the `JSystemContext` and `JUserContext` classes, as they are system or user contexts. Table 2 shows the public methods of the `JSystemContext` class. The `JUserContext` class has the same methods of the `JSystemContext` class, except the `addUser` method.

Class JSystemContext
`JSystemContext(int port)` Creates a new system context associated to the specified TCP/IP port
`void addResource(JResource resource)` Adds the specified resource to the context
`addUser(String name, String password)` Adds the specified user to the context
`void run()` Starts the execution of the context

Table 2: Public methods of class `JSystemContext`

The next program creates a system context, adds two resources to it and then starts its execution.

```
public class ContextLauncher  {

  public static void main(String[] args) {
    JSystemContext context= new JSystemContext(Integer.parseInt(args[0]));
    BufferResource buffer= new BufferResourceImpl("buffer01", ....);
    CalendarResource calendar= new CalendarResourceImpl("calendar2001", ....);
    context.addResource(buffer);
    context.addResource(calendar);
    context.addUser("john", "x2yz");
    context.run();
  }
}
```

In this example, after calling the `run` method the program enters in an infinite loop waiting for containers.

4 Jamp Implementation

Jamp was implemented in Java, using JDK 1.3 and the system has about 2000 lines of code. The basic construction used in the implementation of the system is called *mediator*. A mediator is an internal object used by the system to support container migration and also to support the notion of connected and disconnected references. Every mobile object in Jamp has a mediator, which is created at the same time as its mediated object by the `newObject` method. A mediator has two fields: `guid` and `ref`. The `guid` field stores a value that uniquely identifies the mediated object in any node of the network. This value is the result of the concatenation of the IP address of the machine on which it was created and a value that is unique in this machine across time. The second field of a mediator, called `ref`, is the only reference that exist in the system to a mobile object since the `newObject` method returns a reference to the mediator of the created mobile object and not to the mobile object itself. Figure 2 shows a container from the programmer's point of view and the same container as implemented by the system, with mediators represented by circles.

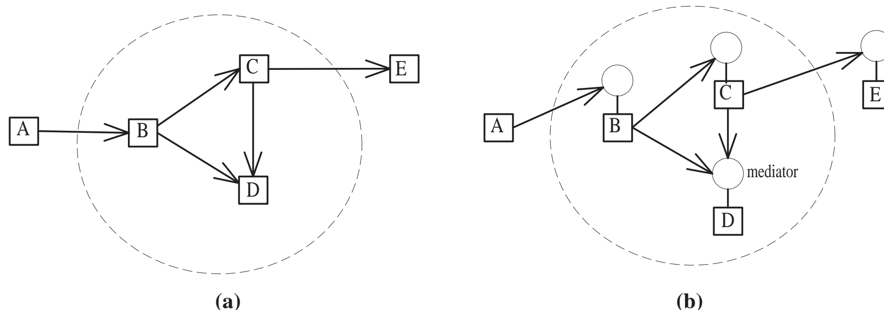

Figure 2: Container in the programmer's view (a) and its internal implementation (b)

Since the `newObject` method returns a reference to the mediator of a mobile object, any call using this reference will first execute a method in the mediator. Therefore, mediators must implement all the methods in the interfaces of their mediated objects. The mediator implementation of these methods should first check the value of the `ref` field. If this field has a non-null value, it should redirect the call to the mobile object. Otherwise, it should raise an exception of type `UnavailableObjectException`. Thus, in the implementation of the system a connected reference is a reference to a mediator that has a non-null mediated object. And a disconnected reference is a reference to a mediator with a null mediated object.

Suppose that we want to send a container $c = \{(m_1, o_1), (m_2, o_2), \ldots, (m_n, o_n)\}$ to a context d, where each o_i is an object of the container and m_i is the mediator of this object. First, we create a serialized representation of the container, using a modified implementation of the serialization mechanism provided in Java. In Java, the serialization of an object p includes the object closure of p, i.e., all the objects that are reachable from p fields. In the implementation of Jamp we change this mechanism in order to restrict the serialization of a container to objects that are included in this container. Basically, the modified version of the serialization mechanism uses the reflection features of the language to void any field in the container that references external objects. After this first step, the method can thus call the default serialization method since the object closure of

the container will not contain anymore external objects.

Suppose now that the destination context d receives the previous container c. First, this context should de-serialize the representation of c. In this process, the destination context uses a class loader different from the default class loader of Java [10]. The class loader used by the system is an instance of the class `JampClassLoader`. Usually, the default class loader retrieves the code of the classes of a program from the file system of the workstation where the program is running. However, in `Jamp` the code of a class can be part of a container and thus we need a customized class loader that can retrieve classes not only from local file systems but also from the serialized representation of containers.

Mediators are also used to check if a container is active or not. A container is active if there is at least one thread running in its objects. In `Jamp`, when a method is called, the mediator of the called object increments a counter of calls in execution in the associated container. Before returning from the call, the mediator decrements this counter. The move method checks this counter and raises an `ActiveContainerException` if it is greater than zero.

Mediators are generated in `Jamp` using the notion of *dynamic proxy class* that is part of the reflection package of JDK 1.3. A dynamic proxy class is a class that implements a set of interfaces specified at run-time [1].

5 Related Work

Mobility is gaining momentum in the design of Internet programming languages. Java [1], for example, has propagated the notion of code mobility in the Internet. Code mobility allows the execution of the same application in different nodes of the network, despite the architecture and operating system of them. But since applets depend on the network to get access to data, the model is not robust to disconnections.

Recently, mobile agents were proposed as an alternative model to the construction of distributed applications in the Internet. A mobile agent is a program that can migrate by the nodes of the network, carrying the state of its execution [17]. Aglets [9], Ajanta [14], D'Agents [6], μCode [11] and JavaSeal [3] are examples of Java mobile agent systems.

In Aglets and Ajanta, mobile agent classes are implemented by inheriting from a pre-defined class that comes with these systems. Since Java does not support multiple inheritance, this solution can restrict reuse of code in mobile agents. Moreover, in both systems, the default Java serialization mechanism is used to transfer an agent and all the objects reachable from it. In Ajanta, after migration, the code of the agent is downloaded on demand from a code base server. Thus, the system is not robust to disconnections. In Aglets, code can be downloaded on demand, but it is also possible to store classes in a JAR file that is transfered along with the agent. This mechanism, however, requires the set of classes to be defined in deployment time and it is not possible to change this set during the agent life. Unlike Aglets and Ajanta, D'Agents provides support to strong mobility using a modified JVM to capture the full control state of an agent.

In μCode, there is an abstraction, called *group*, to define the set of objects and classes that is transfered with an agent. The system, however, does not provide support to communication across groups. Thus communication primitives should be implemented at the application level. JavaSeal also provides an abstraction, called *seal*, to the implementation of mobile agents. Similar to containers, seals have a set of objects and classes. The programmer, however, can not add or

remove classes from seals. In JavaSeal, synchronous message passing via channels is the only inter-agent communication mechanism that exist. Values exchanged over channels are transmitted by deep copy and sharing objects is not allowed across seals. In some mobile applications, this can be inefficient and cumbersome [2]

The notion of containers, contexts and resources were first defined in [15]. In the present paper we describe a system that implements the proposed abstractions in Java. The present paper also introduces the notions of system and user contexts and connected and disconnected references. The style of mobility supported by Jamp is also inspired in some process calculi proposed recently to model mobile computation in the Internet, like the Ambient Calculus [5] and the Seal Calculus [16].

6 Conclusions

This paper has presented a mobile object system, called Jamp, that supports the construction of Internet applications that can run in disconnected mode. There are three abstractions in Jamp to support disconnected operation: containers, contexts and disconnected references.

A container is a group of objects and classes that can be pro-actively shipped to execution contexts provided by nodes of the network. Thus Jamp supports the construction of applications that can be pushed with code and data to their users and thus execute in disconnected mode. This requirement is specially relevant in Internet applications for mobile devices, since wireless networks are susceptible to temporary interruptions of bandwidth and also to voluntary disconnections.

The mobility model used in Jamp organizes the execution contexts of the network into a two-level hierarchy. In this way, the system supports the construction of applications that do not rely on continued connectivity to move and that can execute in several administrative domains. The system also uses the notions of connected and disconnected references to support object sharing without restricting the migration of containers.

Jamp was used to implement part of a conference reviewing system. As future work, we intend to transform containers into protection domains. In this way, we intend to address the security properties that are relevant in any application that executes in an open network like the Internet. We also have plans to implement a cut-down version of the system in the K Virtual Machine (KVM), the VM designed to run in small mobile devices [13].

References

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison Wesley, 3rd edition, 2000.

[2] C. Bryce and C. Razafimahefa. An approach to safe object sharing. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Oct. 2000.

[3] C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, 1999.

[4] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer-Verlag, 1999.

[5] L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[6] R. S. Gray, G. Cybenko, D. Kotz, and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In G. Vigna, editor, *Mobile Agents and Security*, Lecture Notes in Computer Science, pages 154–187. Springer-Verlag, 1998.

[7] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.

[8] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *16th ACM Symposium on Operating Systems Principles*, pages 264–275, 1997.

[9] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.

[10] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. *ACM SIGPLAN Notices*, 33(10):36–44, Oct. 1998.

[11] G. P. Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. In *Proceedings of Mobile Agents: Second International Workshop*, volume 1477 of *Lecture Notes on Computer Science*, pages 160–171. Springer-Verlag, Sept. 1998.

[12] M. Satyanarayanan. Fundamental challenges in mobile computing. In *ACM Symposium on Principles of Distributed Computing*, May 1996.

[13] Sun Microsystems. Java 2 Plataform Micro Edition Technology for Creating Mobile Devices, May 2000.

[14] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. D. Singh. Ajanta - a mobile agent programming system. Technical Report TR98-016 (revised version), Department of Computer Science, University of Minnesota, 1999.

[15] M. T. Valente, R. Bigonha, A. A. Loureiro, and M. Bigonha. Object oriented languages with abstractions for mobile computation. In *Electronic Notes in Theoretical Computer Science*, volume 38. Elsevier Science. (to appear).

[16] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In H. E. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.

[17] J. E. White. Mobile agents. In J. Bradshaw, editor, *Software Agents*, pages 437–472. AAAI Press/MIT Press, 1997.