Disconnected Operation in a Mobile Computation System

Marco T. de O. Valente, Roberto da S. Bigonha,
Mariza A. da S. Bigonha, Antonio A.F. Loureiro
Department of Computer Science
University of Minas Gerais
31270-010 Belo Horizonte, MG, Brazil
+55 31 3499 5860
{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.

Keywords
Disconnected operation, mobile computation, mobile computing.

1 INTRODUCTION

Current advances in wireless networks and portable hardware technology is 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 [11]. 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.

Involuntary disconnections can also happen in the fixed Internet. Recent studies have shown that traditional Web services are unavailable for nearly 15 minutes per day due to failures in the network communication infrastructure [6]. This unavailability can have relevant impacts in applications such as e-business and prevent the use of the Web in mission-critical systems.

Previous research in distributed systems has shown that caching of data is a central idea in disconnected operation [7, 8]. The reason is that cache can be used not only to improve performance but also to enhance data availability. On the other hand, in mobile computing it is important to send to clients not only data but also code. In this way, it is possible to design applications that can be *pushed* to their users with code and data that makes disconnected operation possible. For example, a conference reviewing system can *push* to the program committee members code and data to support reviewing of papers in disconnected mode. Similarly, an e-business site can proactively send applications that allow users to choose off-line the item they want to buy.

In this paper we present the programming model of a system, called Jamp, that uses mobile computation to support operation in disconnected mode. Mobile computation is the notion that the execution of a program need not be tied to a single node of the network [4]. Thus the system described in this paper uses mobile computation (or logical mobility) to deal with the problems raised by physical mobility of portable computing devices. When compared to other mobile systems, Jamp shows differences related to the mobility and communication model supported by the system:

- Mobility model: Jamp has its own abstraction to the implementation of mobile applications, called *container*. A container is a group of objects and classes defined by the programmer. A container is also a mobile entity and thus can move into 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 constrain 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 gives an overview of how Jamp is implemented. Section 4 reviews the related work and Section 5 concludes the paper.

2 PROGRAMMING MODEL

This section presents the mobility and communication constructions that exist in Jamp to deal with the connectivity failures that are typical in the Internet and in wireless networks.

Mobility Model

The main construction available in Jamp to the implementation of mobile applications is called *container*. A container is a group of objects and classes that can be dispatched 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 can be used to support operation in disconnected mode since we can transfer in a single operation 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 and execute containers, fixed nodes or mobile devices should provide a *context* to this execution, i.e., contexts are services 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 wireless networks, 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 option is chosen when the container is shipped to a specific user of a system context. In Jamp, system contexts should run in a node of the fixed network that is always connected to the Internet and it should be possible to send containers from other administrative domains to this node.

A user context is always associated with a system context and a user of the mobile application. From time to time, a user context retrieves all the containers shipped to its user that are stored in the associated system context. The execution of a container in a user context does not start automatically after downloading it. This execution only starts after a request from the user of the context. In Jamp, user contexts can run in mobile devices since they do not need to execute continuously nor need a reliable connection to the Internet.

The system provides support to the so-called *objective migration* of containers [5]. This means that containers in the system can only be moved from the outside of a container. 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 exception is raised. 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 the objects of the container in an inconsistent state.

The mobility model of Jamp is similar to the one used by an electronic mail system, which is the oldest and most popular *push* application available currently in the Internet. In this analogy, a container can be compared to an electronic mail, with the difference that containers have code and data and not only text. A system context can be compared to a mail server and a user context can be compared to a mail reader system.

The mobility model of Jamp can also be compared to the one used by a Java applet [1], which is the most used mobile code application in the Internet. In 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 support both code and data mobility. Containers can also be proactively shipped to users and then 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.

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, as explained below:

- 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.

A connected reference can be used to call methods of the object that it references. But when a method is called using a disconnected reference, an exception is raised. Therefore, the system uses references to name objects, but only a connected reference 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. In general, it is only feasible to support transparent access to remote objects in environments with a low frequency of disconnections, like local area networks [4]. Besides, proxies use non-official TCP/IP ports for remote communication and firewalls usually forbid any traffic through these ports.

A context in Jamp also provides resources to the execution of containers. A resource is a non-mobile object and thus cannot 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 said to be disconnected.

Figure 1 gives an example of how connected and disconnected references work. In that 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, a container is running in the context P and all 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, the case 1 is reestablished.

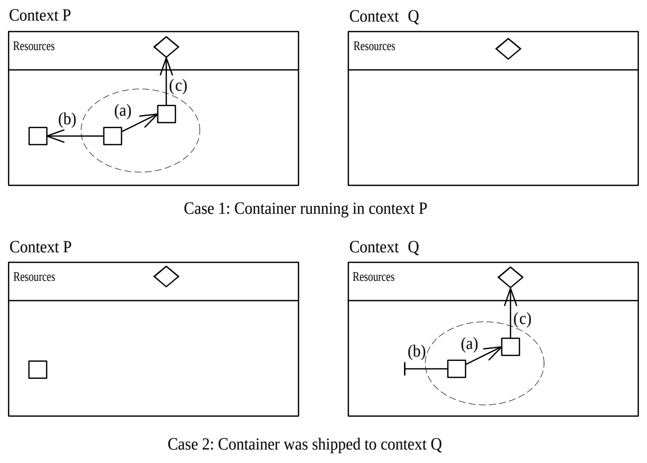

Figure 1: Connected and disconnected references

The communication model of Jamp allows objects in a container to hold references to objects in other containers. Object sharing across containers is an important feature in the design of mobile applications. In case object sharing is not supported, cross container communication should use only copy-by-value semantics, which can be very inefficient. Besides, copy-by-value can make complex the distribution of mutable objects, i.e., objects whose states change continuously, like application environment objects [2]. In Jamp, however, object sharing does create "static bindings" that restrict the migration of containers, since it is possible to change the state of references to external objects when the container move.

3 Jamp IMPLEMENTATION

Jamp was implemented in Java, using JDK (Java Development Kit) version 1.3. The system has about 2000 lines of code. All communication in Jamp is implemented at the socket level and the system does not need any Java RMI support [12]. Java showed to be a good choice to implement a mobile computation system because the language provides support to code mobility, dynamic class loading, object serialization, reflection, multithreading and distributed programming. All these features are presented in the implementation of Jamp.

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 at 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 unique value in that machine across time. The second field of a mediator, called `ref`, is the only reference that exists in the system to a mobile object since the method to create a new object (`newObject`) returns a reference to the mediator of the mobile object created 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.

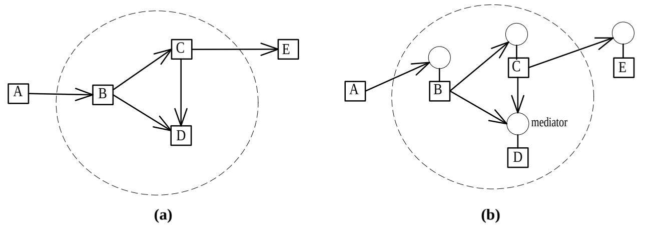

Figure 2: Containers in the programmer's view (a) and its internal implementation (b)

Since the `newObject` method returns a reference to the mediator of a mobile object, any method call using this reference will first execute a method in the mediator. Therefore, mediators must implement all the methods in the interfaces of theirs mediated object. The mediator implementation of these methods should first check the value of the `ref` field. If this field has a non-null value, it can redirect the call to the mobile object. Otherwise, it should raise an exception of the 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 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 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 step, the method can thus call the default serialization method since the object closure of the container will not contain anymore external objects.

When the destination context receives the previous container c it first de-serializes the representation of c. In this process, the destination context uses a class loader different from the default class loader of Java [1]. 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 determine if a container is active or not. A container is active if there is at least one thread running in its object. In Jamp, when a method is called, the mediator of the called object increments a counter of calls being executed in its 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 implements a set of interfaces specified at run-time [1].

4 RELATED WORK

Mobility is gaining momentum in the design of Internet applications. Java [1], for example, has introduced the notion of code mobility in the Internet. Code mobility allows the execution of applications in different nodes of the network, despite architecture and operating system. Java applets, however, depend on the network to get access to data, and thus 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 from node to node in the network, carrying the state of its execution [16]. Aglets [9], Ajanta [14], μCode [10] and JavaSeal [3] are examples of Java mobile agent systems which do not support the programming model proposed in Section 2.

In Aglets and Ajanta, mobile agent classes are implemented using a pre-defined class that comes with these systems. 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 is downloaded on demand or is retrieved from a JAR file transferred along with the agent. This mechanism requires the set of transfered classes to be defined and fixed in configuration time.

In μCode, there is an abstraction, called *group*, to define the set of objects and classes that is transferred 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 [3] also provides an abstraction, called *seal*, to the implementation of mobile agents. Similar to containers, seals have a set of objects and classes but programmers cannot add or remove classes from seals. In JavaSeal, synchronous message passing via channels is the only inter-agent communication mechanism present. Values exchanged over channels are transmitted by deep copy, and sharing objects across seals is not allowed. For some mobile applications, this can be inefficient and cumbersome [2]

The style of mobility supported by Jamp is inspired in some process calculi that were proposed recently to model the notion of mobile computation in the Internet such as the Ambient Calculus [5] and and the Seal Calculus [15].

5 CONCLUSIONS

This paper has presented a mobile computation system, called Jamp, that uses mobile computation to deal with communication failures in wireless networks. The programming constructions available in Jamp can be used to *push* to mobile users code and data that makes disconnected operation possible. The mobility model used in Jamp organizes the execution contexts of the network into a hierarchy. In this way, the system supports the construction of applications that do not depend on continued connectivity to move from a node to another and that can run in several administrative domains. The system also uses the notions of connected and disconnected references to name and access objects across containers.

A formal semantics for the programming model supported by Jamp is under development. This semantics will allow users to prove properties of applications designed in the model. As a case study, Jamp was used to implement part of a conference reviewing system. As future work, we intend to transform containers into protection domains and then address the security issues 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 handheld 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] B. Chandra, M. Dahlin, L. Gao, A.-A. Khoja, A. Nayate, A. Razzaq, and A. Sewani. Resource management for scalable disconnected access to web services. In *Tenth World Wide Web Conference*, May 2001.

[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] G. P. Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. In *Proceedings of Mobile Agents: 2^{nd} International Workshop MA'98*, volume 1477 of *Lecture Notes on Computer Science*, pages 160–171. Springer-Verlag, Sept. 1998.

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

[12] Sun Microsystems. Java Remote Method Invocation Specification, Oct. 1998.

[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] 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.

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