SPRMI: Pure Java Based Shallow Persistent and Distributed Objects

Paulo Rogério da Motta Junior
prmottajr@uol.com.br

Departamento de Informática, Universidade Federal Fluminense
Rua Passo da Pátria, 156 / Bloco E, Niterói, RJ, Brasil
+55 21 8705 8211

Getulio Moreira
getuliogsm@hotmail.com

Maria Alice Brito
malice@ime.uerj.br

Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro
Rua São Francisco Xavier, 524 / Bloco B – sala 6020, CEP: 20.550 – 013, Rio de Janeiro, RJ, Brasil

Abstract

In this paper, we present SPRMI, a Java package to support shallow persistent and distributed objects in a lightweight fashion. Its main characteristic is the shallow object instantiation capability. In other words, at the moment an object is instantiated, objects referenced by its attributes are not instantiated. This is achieved by the use of dynamic proxies as reference to objects of non primitive data types. These proxies use an identification strategy based on primitive data types that prevents direct references to target objects. Therefore, the serialization mechanism, when accessing remote or persistent objects, does not occur in a transitive way, because it is not possible to propagate through the object's composition graph. Three benefits can be derived from this ability: 1) parameters may be passed in a call-by-reference manner, during the invocation to a remote object, becoming closer to local parameter passing semantics; 2) the use of dynamic proxies drives the application towards transparency of persistence and distribution; and 3) it is possible to execute applications with large number of objects.

Keywords

Java RMI, Shallow Persistence and Distribution, Proxy.

1. Introduction

Considering the programming languages construction as our context, we have developed a simple library to add shallow instantiation to the read/write primitives and local/remote object access, available respectively on Object Input/Output Stream [1] and RMI [2] of Java's basic API.

In these primitives, serialization is used to support objects encoding, and the objects reachable from them, into a stream of bytes; and it supports the complementary reconstruction of the object graph from the stream [3]. This mechanism is an implementation of transitive persistence, the third principle of orthogonal persistence [4], stating that an object persists to execution if it is a persistence root or reachable from a root, by the references, recursively.

This serialization mechanism is important and useful, however critics of the undesired side effects caused by it are not recent, for example, the performance inhibitor factor, with the use of Java RMI. The solutions for each of these factors, appointed on [5, 6, 7, 8, 9], tries to reduce the effects of the recursive process executed for object encoding and reconstruction. Besides that, risks arise from direct and indirect reachability from a remote, persistent object [10, 11].

The solution, adding an indirection level on the object reference by the use of a proxy, is encouraged by the discussions found in [12, 13]. In [12], the authors make two choices, among others: 1) the use of a proxy to reference remote objects, and 2) the "instance methods execute on the node where the object was allocated, and not on the node where the reference is made". On this way, structural solutions that extend Java RMI appeared, for example, the combination of smart proxy with interceptor [14], implemented by the Dynamic Proxy API [15], or the solution with fragmented-object model [16], that is similar to the smart proxy solution. By these means, serialization may be avoided.

Beyond these serialization effects, there are also some critics about the diversity of parameter passing semantics, defined for Java and Java RMI. Remembering that, in case of the formal parameter is declared with user-defined type, the real parameter is passed by reference on the local call. If it is a remote call and the real parameter is an instance of a class that implements the Remote interface, it is made by reference, but if the parameter class implements the `Serializable` interface instead of `Remote`, the call is made by value, with a copy of the parameter. With `Array`, local call is made by reference,

and, differently, remote call, since array is `Serializable` by default, it is made by value with a copy of the array. The discussions, brought on [17, 18, 19, 11, 8, 6], are about the semantics of parameter passing. Initially, it was specified for the language, as by value (semantic mode in), and its conflicting differences described above, making it uncertain to the programmer of which one was used on the client side. Although, by the language specification, the parameter passing definition is by value, the differences on the semantics described above are of public domain and the programmer is aware of the formal parameter type.

About the risks of operating with persistence/distribution and overhead factors, in Java RMI, brought by serialization, the solutions for each of these followed two alternative paths: 1) the use of the recursive process executed for object encoding and reconstruction, or 2) the use of a proxy to reference remote objects, to avoid passing through serialization, in that each target object method is executed on the node that it resides on.

On the persistence of distributed objects, that are present on some composition graph, when the structure of a proxy contains only primitive type attributes, the risks pointed in [10] do not rise. The proxy does not allow serialization to propagate throughout the composition path. In this way, each object may always be instantiated and persisted on its local residence, and, being this object an invocation target, the corresponding method is processed where this object lives [12].

Still with the proxy, the performance aspect may improve or get worse, depending on how much the objects are distributed on the composition graph of the target object and the parameters passed by reference. This is because, respectively, on one hand, the processing of each remote call is freed of the overhead caused by Serialization, but, on the other hand, it may appear some loss due to the increasing network traffic because of the many remote calls that may happen during the execution of the invoked remote method [11].

Beyond the solutions presented above, the use of the proxy, with an object reference, may collaborate to the standardization of the parameter passing semantics, on the polemic case of formal reference types, becoming by-reference, both on local and remote calls.

These opportunities brought by the proxy, when used as user-defined object references, lead us to the choice of the structural solution, similar to the combination of the smart proxy with interceptor [14], in which we also use the Dynamic Proxy API, adding the persistence facility. By our solution, we define the instantiation of an object in the shallow mode. The class to the proxy instance implements the `Serializable` interface and its structure is based on primitive data types, to block Serialization recursion, on the object that is instantiated on-demand. Such a structure is reached, simply by filling the user-defined attributes with these proxies, causing one more indirection on the reference. This structure affects the mechanisms of both persistence, in Java Object Input/Output Stream primitives, and (un)marshalling, in Java RMI.

If objects are structured in this shallow mode, each proxy-referenced object will only be instantiated on the invocation time, playing the role of invocation target. If, on the method execution of this invocation, other invocations to other target objects take place, these, will be instantiated, at the invocation time and so forth. These objects may be on the local or some remote machine, or even not been yet instantiated, due to being persistent. The attributes of primitive type are filled as on traditional Java programming.

The shallow representation of these user-defined objects contributes to the standardization of the parameter passing semantics, when the formal parameter is of a user-defined type, making it in-out, in a fashion similar to call-by-reference on the local and remote calls. With this, the call-by-copy does not happen anymore, reducing the quantity of data transmitted on the remote call communication. This decrease in the amount of data helps to explain the better performance achieved by SPRMI, when compared to Java RMI, on the tests presented on section 3, balancing the loss with the increase in network traffic in expectation above.

On the same time, the use of a dynamic proxy, as an indirect reference to persistent/remote objects, brings the opportunity to hide the persistence and distribution aspects from the application code, making these aspects transparent to the application programming.

The choice for Object Input/Output Stream characterize our model as distributed lightweight persistence [20, 21], differently from the distributed heavy-weight database systems, that offer a lot of features, such as query management and consistency management. According to the authors, in [21], these lightweight models are needed for many applications at hand.

The rest of this paper is organized as follows. In section 2, we present our persistency and remote access strategies, and also our architecture model together with its elements collaboration. In section 3, we bring an example and the application setup, to present how the library is used; and a performance comparison among original Java and our model, by some test results of remote calls parameters making references to objects which composition graph presents some density. In section 4 we describe related works. Section 5 concludes the paper and highlights future questions and works.

2. Programming Model

Our programming model is based on a simple persistence and distribution library that allows users to easily develop distributed and persistent applications by referencing objects through Dynamic Proxy API. These proxies use an `Identifier` class to hold information about object identification and location. Target objects are stored in simple object storages, implemented by a Hash Table, being the brokers the intermediary to access

these storages on their hosts. The proxies, used to reference target objects, communicate with brokers through traditional RMI infrastructure to deliver method calls. The elements that compound our architecture are explained bellow, together with the description of the collaboration among them, so that the persistence and the arrangements for remote object method invocation work.

2.1. Architecture Overview

In order to incorporate persistence and distribution, the application needs to import SPRMI package and use the `ShallowProxy` class to create the references to objects. Although our package uses the traditional RMI infrastructure for communication, there is no need for objects both to be compiled with *rmic* and to be published on the *rmiregistry*. Instead, we developed a broker component that combined with the object persistent storage is responsible for object persistence and performing the method invocation on application target object. Figure 1 shows the Architecture representation with two hosts. Communication between objects takes place through the RMI connection that is established from a proxy to a broker and method calls are serialized, by reflection, in order to be sent.

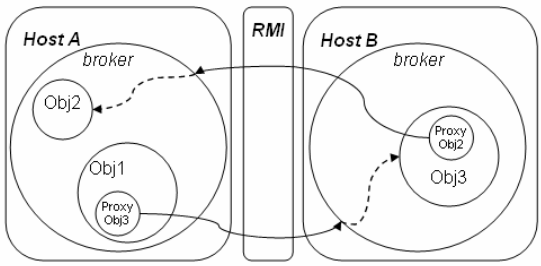

Figure 1. Architecture representation. Communication from a proxy to a broker

Each object created by an application is accessed through a dynamic proxy and resides, on the persistent object storage, which is referenced by a broker attribute. The proxy is specified at instantiation time by a constructor call. Both objects and brokers are identified by unique system-wide numeric id that is requested at creation time, once assigned this id will not change and can be used to allow executions to resume from a persistent state.

2.2. Prototype Considerations

For the prototype that we developed, as a proof of concept, some issues were resolved in a naïve implementation and left for further improvement. These items are:

- System-wide Identifier Store – considering that objects and brokers need to be uniquely identified throughout the system, it was necessary to have a way to generate unique identifiers based on some rule. For this end, we created a special entity that acts as an identifier store. It is accessed only inside the brokers. Although being already thread safe, this is a centralized entity that has to become distributed in the future.
- Central Host – this is a virtual entity that holds the table of brokers that are part of the infrastructure in use. This role is assigned to a chosen broker and its network address and port are informed to other brokers at start-up. The importance of this entity is related to the ability to find other brokers. This centralized approach is recognized by us as a bottleneck for scalability and will have to be redesigned to become distributed in the future.
- Broker checkpoints – an elaborated checkpoint strategy for broker persistence is out of the scope of this prototype. In order to maintain broker state properly, we choose to call the broker `write` method periodically. This basic approach allows infrastructure consistency. A note on broker persistence, it is based on the traditional Object Input/Output Stream [1] and not using SPRMI mechanism, by a natural reason, this is on the SPRMI implementation level.

2.3. Architecture Terms

The terms used for our architecture are as follows: 1) Target object – application object that is the destination of a method call; 2) Caller object – application object that is the origin of a method call; 3) Remote object – any application object that resides in a remote location regarding a certain broker. Remote objects are accessed via broker infrastructure, see bellow; 4) Immediate state – object's set of primitive data type fields representing its real state; and 5) Referential state – object's set of non-primitive data types fields that represents references to other objects. References are made based on unique system-wide numeric identifiers. Non-primitive data types are also called user-defined types.

Broker

Our broker component, that implements the message broker pattern, is a RMI enabled class that is accessible through the *rmiregistry* service. It allows serialized method calls received from caller objects to be invoked on target objects that reside in its persistent object storage and returns the result of method calls, when present. Both caller and target objects can be local or remote. In SPRMI, method calls are serialized and transmitted through the network, once received by the broker, they are reconstructed by reflection mechanisms and effectively invoked on the target object. Each broker is identified by a unique system-wide numeric identifier. This identifier is used at run time to find and reference a broker to accomplish communication. The broker infrastructure uses a table of broker addresses indexed by broker identifiers. The process of broker search is encapsulated in a static method called `findBroker` that

is implemented on the `Broker` class.
The object store, accessed by the broker, relies on a Hash Table, which entries are of `TableEntry` type exclusive to each object stored, described ahead.

Dynamic Proxies: Invocation Handler

In SPRMI, objects are not directly referenced. To achieve our goal of shallow persistence it was necessary to break down the composition graph that Java creates for each object that declares fields of non-primitive data types. For that, we introduced a special proxy class that holds information about object identification and location, regarding the broker infrastructure. This information makes it possible for the proxy to communicate with the broker, where the target object resides in, to send serialized method calls. Using the `findBroker` method, described in the last section, a reference to the remote broker is received. With the broker reference at hand, it is possible to request that a method be invoked on a certain target object, which is known by the proxy through the object id that it keeps. Figure 2 presents how the object representation is modified by using dynamic proxies, consequently, causing one more level of indirection on the object references belonging to the composition graph.

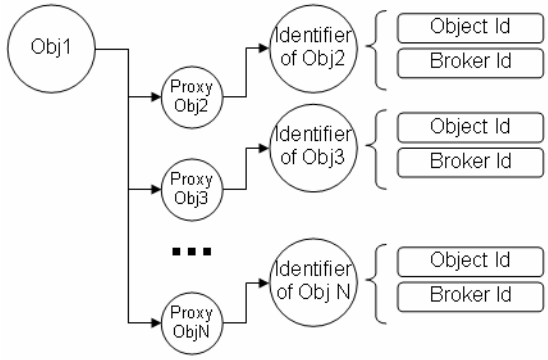

Figure 2. Object composition graph using dynamic proxies

As we can see on Figure 2, when using dynamic proxies to reference target objects it is possible to overcome the deep composition that is used by the Java system. When an object is persisted with SPRMI only its immediate state gets saved, by immediate state we refer to object's fields that are of primitive data types. We can think of fields that are of non-primitive data types as independent objects that have their own immediate state, and object persistence should not force referenced objects to be persisted. Instead, we should make persistence only the identification that allows us to reference those objects at run time and we do so by saving together, with the objects immediate state, all the proxy objects that holds the information to other objects. These effects characterize our conception as a structural model and lead us to make use of the shallow object instantiation solution.

Object Persistent Storage and Table Entry

The object persistent storage is implemented by the generic `HashTable` class, with the `TableEntry` class as its entries. We should remember that this reference is kept by an attribute on the broker.

An instance of `TableEntry` may be briefly called entry. It maintains the target object reference, the name of the file where the object resides, the class of the object and the identification of the object.

When an application object is created, the dynamic proxy requests the broker to provide the creation of this object. On this action, an entry is created and included on the persistent storage, its attributes should be filled and the object must be persisted for the first time. At the end of the creation process on the broker, the object id is returned, and is kept in the proxy together with the broker id.

The `TableEntry` class implements `read` and `write` methods that are responsible for individual object persistence, for that, a field keeps the file name. Other special method is the `runMethod` that is responsible for actually invoking methods on the target object. At this point, a verification is made to ensure that the target object is instantiated, if not, it is read from persistent storage to be instantiated, its reference is fulfilled, and the method then is, finally invoked. Once instantiated, the object remains in main memory. This measure produces the effect of on-demand object instantiation.

Identifier

The identifier is a simple class that has the object and broker identification and is used by dynamic proxies to find a reference to the broker, and by the broker's object persistent storage, to find an object on storage. Another important role played by an instance of this class is to block the persistence recursion, due to its structure having primitive data type values. These identifiers are used on proxies. In its turn, in case of both local variables, parameters or attributes are declared with user-defined type, these proxies are used on them.

Method calls

The Java RMI method invocation specification expects that we reference the target object directly. From the application programmer's perspective, there is no syntactic difference, but, with SPRMI mechanisms, the method call is serialized inside the proxies and sent to the broker where the target object resides in its object store. As explained, the broker will retransmit the method to the storage's entry, which will reconstruct the method based on the target object's reference, thus the invocation can take effect. We explain these processes bellow.

Serializing method calls

With SPRMI, the marshalling process is applied to the method, which is sent to be invoked by the broker. Once a caller object tries to process an `invoke` method, the proxy intercepts this invocation and passes it to the handler `invoke` method. The expected parameters for this method have to be as follows: a reference to the proxy object itself, a reference to the method that has to be executed on the target object and an array of the

`Object` type, which holds the parameters to the target method. Since a reference to the target method can not be serialized to be sent over the network, we created a special class, `RunMethodParameters`, which instances holds: the method name, its argument array and the target object id. Such instance has all the requested information for the broker to reconstruct the method reference and perform the call.

The proxy uses the broker identification maintained on the identifier object to find a reference to the target broker, which is accomplished through the `findBroker` method against the Central Host. Once this reference is returned the broker's `runMethod` is executed passing the information for method reconstruction.

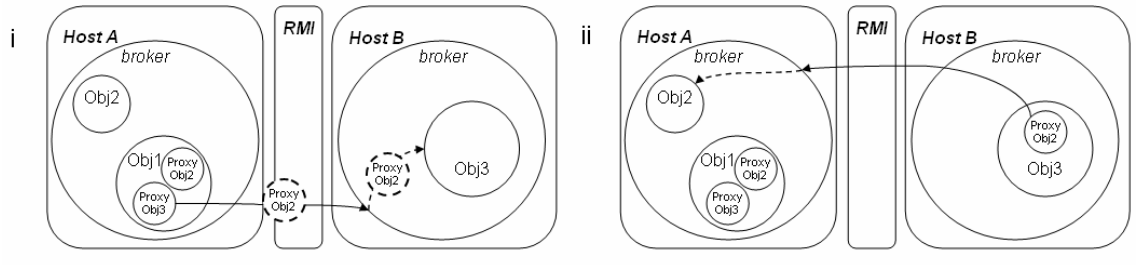

Figure 3. Representation of argument passing in SPRMI. In (i) the proxy is sent to the target object. In (ii) the proxy is used to call back referenced the object.

Reconstructing method calls

The `runMethod` implemented on the broker is exposed via the traditional RMI mechanism making it possible for proxies to execute it. This method receives the serialized information and rebuilds it on its original pieces, in order to call the `runMethod`, on the correspondent target object entry. It is worthy to note that both the broker and the entry has an instance method called `runMethod`, and the course is always first by the broker and then by the entry. On the entry, the `runMethod` receives the parameters: the method name and its arguments array, being these parameters used to recreate a call to the correct method. Using reflection mechanisms, we iterate through the arguments array to find its classes. This step is necessary due can exist another method with the same name, which difference relies on the parameter list. Once identified the proper method, it is invoked to the target object that is referenced by the entry. As explained earlier, if the target object is not instantiated in main memory, before method invocation, it has to be restored from permanent storage. Once the method is executed the return value is sent back to the client code.

Argument passing

When a method needs arguments for its execution, the `invoke` method of the corresponding proxy receives an array containing all the data that needs to be serialized and sent to the target object. Among this data, other proxies may be sent as arguments and will be reconstructed on the target object's side. Once reconstructed, these proxies can be used by the target object's methods to invoke methods on the objects referenced by them, and so on.

Figure 3 represents an example scenario of this situation. Let's consider two hosts *A* and *B*, a set of three objects called *Obj1*, *Obj2* and *Obj3*. Using proxy references, *Obj1* can reach *Obj2* and *Obj3* and through the proxy a method call is sent to *Obj3* passing *Obj2*'s proxy, as parameter. This call is shown on Figure 3-i, at left hand side. Once received the method call is reconstructed as explained above, and *Obj3* receives a proxy that contains the identification that allows it to send a method call to *Obj2* on the correct broker. This last situation is shown on Figure 3-ii, at right hand side.

3. Experiments (Usage and Evaluation)

From usage perspective, we consider our primary goal to provide programmers with a more flexible and easy-to-use tool to achieve application persistence and distribution. From quantitative point of view, we created a set of tests that compare two similar applications, using SPRMI and traditional RMI.

Traditional RMI applications require the programmer to compile the *server* classes with a special compiler provided by Sun together with the JDK. Besides that, it is also necessary to implement some *lookup* techniques to bind the client and server together. If an object is to be sent from client to server, it *must* implement the `Serializable` interface. For SPRMI applications, no special compilation is necessary, however the `Serializable` interface constraint still holds. To make use of SPRMI capabilities, an interface for each user-defined type has to be created and all references to instances of these types must be made through these interfaces. The last requirement is to use a special method for object instantiation.

Let consider, for example a user-defined `Book class`, it must implement the `Serializable` and a `BookInt interface`. The instantiation is as follows:

```
BrokerInterface b =
  Broker.findBroker("[ip
address]");
BookInt obj =(BookInt)
ShallowProxy.newInstance(b, new
Book());
```

Although the object instantiation is modified to accomplish SPRMI initialization, the method invocation syntax of a local and transient target object is the same as for a remote and/or persistent one. Our advance, compared to Java's Basic Persistence, is transparency, meaning that reads and writes and distribution aspects were hidden by the Dynamic Proxy and other intermediaries [22].

3.1. Experiments description

The example application is a simple program which connects to a server object. It executes a remote method passing a local object reference as a parameter. The main goal for this experiment is to validate the behaviour, when the parameter grows in size *and* object composition in depth. There are 4 steps on the experiment presented on figure 4 bellow.

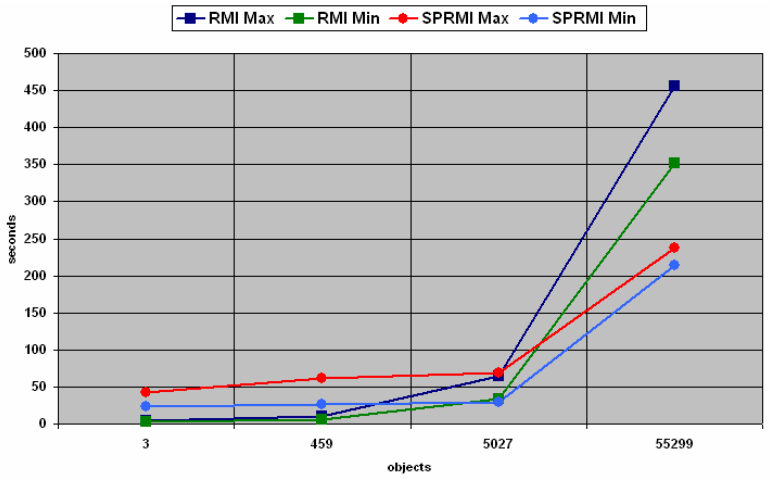

Figure 4. Performance comparison between SPRMI and traditional RMI

The figure above represents both maximum and minimum length of time for each technology on each step of the experiment. Each step was collected through a thousand calls to the remote method that receives the object argument. For the first step the argument was a simple object with three attributes of the String type and no references to other user data types. For the second step, composition depth was increased to level 2, with an attribute of a user data type, both technologies remain almost constant. For the third step, composition depth was increased again for 3 levels of user data type references, the number of objects increased by a factor around 10. On the third step traditional RMI technology gets a little above SPRMI, even though RMI is not carrying any persistence overhead. On the fourth and last step, composition depth was increased to level 4 and the number of objects again increased by a factor around 10. For this last one, RMI time grows dramatically due to the size of the argument that is transmitted, however SPRMI grows by half the time RMI does even for a large number of objects. This can be accomplished by the reference through Dynamic Proxies, when transmitted over the network, only the remote objects ids are sent saving network bandwidth.

3.2. Experiments setup

Considering that brokers will be used as server objects they must be started prior to any application in order to establish the necessary communication infrastructure for the application's objects.
For our experiment we used a simple scenario using two distinct brokers. In order to simplify the start-up process, script files were developed allowing the easy execution of many test runs. For the broker to start executing it needs to receive four parameters: 1) its *rmiregistry* host, 2) its port, 3) the file name that will be used for its persistence and 4) the central host ip address. With these parameters the brokers will start by trying to resume the last run if the file exists, if not, they will reference the System Identifier Store in order to receive a valid unique id. Once the id is received, they register at the central host informing their id, ip address and port. After this step, they cycle through the checkpoint routine while waiting for requests from caller objects.

4. Related Works

Serialization has been considered a performance inhibitor factor, with the use of RMI, with overhead identified, as for example: the need to generate information that would need to be propagated through references [5]; the very slow way to examine and update unknown objects, in Java reflection [6]; the need to (un)serialize [7]; RMI supports polymorphism, which requires the system to be able to process type information at run time and download remote classes into a running application [8]; RMI is designed for wide-area and high-latency networks, it is based on a slow object serialization, and it does not support high- performance communication networks [9]. The solutions for each of these factors appointed try to improve the recursive process executed for object encoding and rebuilding in RMI.

In the context of persistence with distributed objects, in addition to the performance loss, other problems may arise when we try reachability in graphs that have local and remote objects, as demonstrated by Susan Spence in PJRMI [10]. In this work, a support for remote method invocation in the context of the object-oriented, orthogonally-persistent system of the PJama Project, when there is an object on the graph that was not created to be persistent: 1) it will become persistent by reachability; and 2) in case it does become persistent, in which machine will it reside. These difficulties demanded a special support in PJRMI, to handle all the three situations: 1) Detecting no persistence by reachability; 2) Determining non-persistence of remotely target objects; and 3) Supporting the movement of stores between hosts. This approach was inserted in a Java RMI module called Distributed Garbage Collection.

Another aspect in this context is the transparency of the persistence and distribution functionalities, that has been resolved with proxies [12, 13, 22, 14], allowing that programming do not need to distinguish the references between objects without/with special features added, like remote or persistent conditions. A reference to an object with additional features is represented by a *proxy* object that contains attributes, according to the functionality that it represents.

The use of proxy results in one more indirection level on the object's reference, what causes some resistance to its adoption. With this issue in sight, it is important to highlight some comments found in [12, 13], that encourage its use. The two choices adopted, in [12]: 1) the use of proxy to reference remote objects, and 2) the "instance methods execute on the node where the object was allocated, and not on the node where the reference is made". The arguments presented, in [13], about the Dynamic Proxy use created for an interface `I`, as a "consistent manner wherever an object of that type or any descendant type is expected". These solutions with proxy influence the object structure, and this is the reason why they are called structural.

In [14, 16], that are Java RMI extensions, the concept of proxy helps the distribution of the object composition parts. In [14], the structural model was achieved by the addition of smart stubs to RMI, being necessary an indirection level both on client and server. This stub is implemented by the Dynamic Proxy API. FORMI [16] is a RMI extension to support a flexible fragmented-object model, which reminds the smart proxy.

These last two works are the most related to ours. They added facilities, without the client being aware, of the proxy or fragment. Our solution also includes the facilities under the proxy and is implemented by the Dynamic Proxy API, combined with a simple Broker, that, counts with and also simple object persistent storage, implemented via a Hash Table. The broker is a framework, that relies on Java RMI for communication, but that can be extended to use other communication means.

Our solution is very simple and is characterized as structural. It is achieved by the shallow object instantiation, with user-defined attributes being filled with a proxy, which references objects by primitive type identifiers. With this, persistence by reachability is blocked, that means, the transitive persistence, the third principle of orthogonal persistence [4], is absent in our work. The instantiation of an object, either local or remote, will always occur on demand. Our work is pure Java with no changes, be it on the language, the object, the compiler or the JVM, relying on Serialization mechanisms and being lightweight (without consistency and query features) [21].

We bring the different parameter passing semantics question, because the proxy helps the standardization, in the case of the user-defined formal parameter. In [19], is presented a new form of parameter passing, adding a copy clause to force the parameter copy of user-defined types, also called reference-type. In [18], CORBA and Java RMI are questioned, by the programmer point of view. In CORBA, by the need to use interfaces and, in RMI, by the need to implement the Remote interface, on the object class, forcing an anticipated decision of which objects should be local or remote. They present a library of Reflective Remote Method Invocation classes. This library supports Java RMI facilities, but is not tied to the JVM, and makes use of the ability to call methods using reflection, eliminating the need for a `RemoteObject` and stub/skeleton generation and thus make it possible for any object to be served and used remotely. With the intent to turning the parameter passing into reference-type, on the remote

call, equal to the local call, the authors, in NRMI [6], achieved a modified version of the Java RMI. The complications of copy-restore, mainly, caused by aliasing references, were treated in an algorithm, that access all the reachable objects from the reference type real parameter. In NRMI, the programmer may select the call-by-copy-restore semantic, declaring a class to implement the Restorable interface.

By our structural solution, in a local/remote call, when the real parameter is associated to a formal parameter of the user-defined type and is referenced by a proxy, the parameter passing semantics has the same effects as the Java local call, i.e., in out semantics mode. The differences on the parameter passing semantics for the Array type could not be resolved, because there are internal aspects out of our solution's control. The original parameter passing for primitive types already has preserved the same semantics for the local/remote calls.

In the works above, there were also questions of where the solutions should be located, in a library, on the compiler, on the virtual machine and even on the communication protocols. Our work is a simple library that combines the facilities of object persistence and distribution, by use of RMI and Object Input/Output Stream primitives, reducing the effects of Java Object Serialization, with a structural solution via shallow instantiation of objects. This library does not offer facilities for consistency and query, characterizing, this way, as lightweight persistence.

5. Conclusions

Our work was lead by the structural solution, with the use of a proxy, implemented by the Dynamic Proxy API. The main characteristic of our model is the shallow instantiation of objects, achieved by the use of proxies on the user-defined type attributes, local variables and parameters that, on its turn, contains primitive type objects. On this way, the instantiation of an object does not cause the instantiation of the objects on its composition graph.
This model brought the following contributions: 1) avoids the problems with distributed and persistent objects, accessible by reachability, properly illustrated, in [10]; 2) the standardization of the parameter passing semantics, when the formal parameter is of a user-defined type, in which the semantic is the same as the original Java local call, i.e, the parameter passing becomes call-by-reference. In this case, the real parameters are not anymore passed by copy; 3) this absence of copy reduces the amount of data on the communication of each remote call, improving performance. This happens even in composition graphs with greater distribution density, that cause an increase of remote calls, as can be seen by the results of the tests presented on section 3; 4) the transparency of persistence aspects compared to the Object Input Output Stream API; 5) if the objects can be on-demand instantiated, it becomes possible to execute applications with large numbers of objects.
Another aspect of our library is the lack of features such as query management and consistency management with access to distributed storage, which gives it the lightweight distributed characteristic [20, 21], being essential to avoid redundancy of functionalities of recovery and concurrency control in the protocols that we intend to extend.

As future works, the broker that we implement needs to resolve garbage collection of objects that are no longer referenced. Other future work is to include concurrency control protocols and asynchronous mode calls, with *Future*, in a transparent fashion.

References

[1] Sun Microsystems, "Object Input/Output Stream Java Object Serialization Specification", http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/.

[2] Sun Microsystems, "Sun Java Remote Method Invocation Specification", http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html.

[3] Sun Microsystems, "Object Serialization", http://java.sun.com/j2se/1.4.2/docs/guide/serialization/ .

[4] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.. (1983) "An Approach to Persistent Programming", Computer Journal 26, 4 pp 360-365.

[5] Breg, F., Polychronopoulos, C. D. (2001) "Java Virtual Machine support for object serialization" In *Special Issue: ACM 2001 Java Grande-ISCOPE (JGI2001) Conference*. Geoffrey C. Fox., Illinois.

[6] Tilevich, E., Smaragdakis, Y. (2003) "NRMI: natural and efficient middleware", In *Proceedings. 23rd International Conference of Distributed Computing Systems*. Atlanta. p. 449-460.

[7] Kono, K., Masuda, T.(2000) "Efficient RMI : Dynamic Specialization of Object Serialization", In: *Proceedings. 20th International Conference on Distributed Computing Systems*. Taiwan. pp 308-315.

[8] Maassen, J., Nieuwpoort, R., Veldema, R., Bal, H. E., Plaat, A. (1999) "An efficient implementation of Java's remote method invocation" In: *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM Press, New York. pp 173 – 182.

[9] Nester, C., Philippsen, M., Haumache, B. (1999) "A more efficient RMI for Java", In: *Proceedings of the ACM 1999 conference on Java Grande*. ACM Press, New York. pp 152 – 159.

[10] Spence, S. (1999) "PJRMI: Remote Method Invocation for a Persistent System", In: *Proceedings of the International Symposium on Distributed Objects and Applications*. Edinburgh, United Kingdom.

[11] Spence, S. "Policies for Passing Objects by Copy between Widely Distributed Persistent Stores", *Regular Presentation to PODC 2000*. University of Glasgow, Glasgow, Scotland, UK.

[12] Hicks, M., Jaganhattan, S., Kelsey, R., Moore, J. and Ungureanu, C. (1999) "Transparent Communication for Distributed Objects in Java", In: *Proceedings of the ACM 1999 Java Grande Conference*. ACM Press, New York.

[13] Eugster, P., Baehni, S. (2002) "Abstracting Remote Object Interaction in a Peer-2-Peer Environment", In: *Proceedings of the ACM 2002 Java Grande Conference*, ACM Press, New York.

[14] Santos, H., Marques, P., Silva, L. (2002) "A Framework for Smart Proxies and Interceptors in RMI", In: *Proceedings of The 15th International Conference on Parallel and Distributed Computing Systems (PDCS - 2002)*. Louisville, KY, USA.

[15] Microsystems., "Dynamic Proxy Classes", http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

[16] Kapitza, R., Kirstein, M., Schmidt, H., Hauck, F. (2005) "FORMI: An RMI Extension for Adaptive Applications", In: (*RM´05)* The *4th Workshop on Reflectives and Adaptive Middleware Systems of ACM International Conference Proceedings Series* Grenoble, France.

[17] Hof, M. (1999) "Object Model with Exchangeable Invocation Semantics", In: ECOOP Workshop for Phd Students in OO Systems.

[18] Thiruvathukal, G., Thomas, L., Korczynski, A. (1998) "Reflective Remote Method Invocation", Concurrency: Practice and Experience, 10 (11-13): 911-626, September-November.

[19] Brose, G., Löhr, K., Spiegel, A. (1997) "Java Does Not Distribute", In: Proceedings of Technology of Object Oriented Language and Systems. TOOLS Pacific´97, Melbourne, Australia, November.

[20] Kappel, G., Schröder, B. (1998) "Light-Weight Persistence in Java - A Tour on RMI- and CORBA-Based Solutions", In: *Proceedings of the Database and Expert Systems Applications: 9th International Conference (DEXA'98)* Vienna, Austria, August. Springer, Berlin.

[21] A., Kamil, A. (2005) "pobj: A Lightweight Persistent Objects Library and Its Application to Persistency in Titanium/Java", http://www.eecs.berkeley.edu/~arnold/coursework/cs262a/pobj.pdf. January.

[22] Paal, S., Kammüller, R., Freisleben, B. (2003) "Java Remote Object Binding with Method Streaming", In: *Proceedings of the 4th International Conference for Objects, Components, Architectures, Services and Applications for a Networked World (NODE 2003)*. Erfurt, Germany, S. p. 230-244.