A Framework for Smart Proxies and Interceptors in RMI

Nuno Santos Paulo Marques Luis Silva
CISUC, University of Coimbra, Coimbra, Portugal
{nsantos, pmarques, luis}@dei.uc.pt

Abstract

The Java Remote Method Invocation (RMI) API shields the developer from the details of distributed programming, allowing him to concentrate on application specific code. But to perform some operations that are orthogonal to the application, like logging, auditing, caching, QoS, fault tolerance, and security, sometimes it is necessary to customize the default behavior of the RMI runtime. Other middleware for distributed programming, like CORBA and the Remoting framework of the .NET platform, support smart proxies and interceptors, which can be used for these purposes, allowing the separation of application-specific code from service-specific code. In RMI there is no direct way of doing so. This paper presents a framework based on the Dynamic Proxy API for using smart proxies and interceptors with RMI. This framework requires no changes in the client application and minimal changes in the server application, giving the developer greater control over the distributed application. A practical example of use is also given, by using the described framework to implement user authentication and fine-grained access control in RMI.

Keywords : Java RMI, Smart Proxies, Interceptors, Security

1 Introduction

The Java Remote Method Invocation (RMI) API [1] allows programmers to build distributed applications at an high level of abstraction. The programmer needs only to concentrate on the application-specific code, and build the interfaces and objects that belong to the semantic space of the application. The code necessary to handle the distribution of objects and the communication between the client and server, is generated automatically from the interfaces, by tools like the `rmic` compiler.

This is certainly a big advantage as it brings distributed programming to the average programmer. But sometimes it is necessary to change the default behavior of Java RMI at layers deeper than the application layer. This is common when implementing functionality like logging and accounting, client-side caching, QoS and fault tolerance, or security. Common to all these situations is the presence of two distinct layers of functionality. There is application-specific code and service-specific code. The service-specific code is not specific to a particular application, and needs no knowledge of the application that is being used. The same functionality may be used with different applications. RMI gives little choice other to implement this type of service-specific code at the same level of the application code. This burdens the programmer, reduces modularity and leads to complex designs.

There are two concepts which may be used to address these issues: *smart proxies* and *interceptors*.

A *smart proxy* [2, 3] is like a stub that can perform more tasks than simply forwarding method invocations to the server. The developer defines a smart proxy that is sent to the client in place of the stub. This way, the developer has an hook to implement service-specific code which is executed at the client side without being necessary to change the client application. This is transparent to the client, which receives an object that implements the remote interface that was requested, and needs not to be concerned with what really happens in that object, as long as it acts like it is specified in the remote interface.

An *interceptor* [3] is an object invoked by the communication system in the path of the remote method invocation. It can be present both at the client and at the server, and can be invoked before the client sends the request to the server, before and after the server executes the request and after the client receives the response.

Together, these notions allow the developer to build an elegant solution to the problems presented before.

This paper demonstrates how these notions may be integrated in Java RMI with little effort. The Dynamic Proxy API [4] is used for creating smart proxies and support interceptors. Applications can use these notions with minimal changes at the server side code and no changes at the client application. A practical example is provided, where this approach is used to implement user-authentication and fine-grained access control in RMI. This is a feature we believe is lacking in RMI, especially since the introduction of the Java Authentication and Authorization Service API [5], which introduces the notion of user on the Java framework, allowing access-control to be based also on who is executing the code. Unfortunately, in a remote method invocation the information about the user is lost, and the server has no way to implement access control based on the user. We demonstrate how it is possible to overcome this limitation, by using smart proxies and interceptors to extend the JAAS model to RMI.

The remainder of this paper is organized as follow: Section 2 describes the proposed approach. Section 3 applies the proposed framework to the implementation of security in RMI calls. In Section 4 we present the results of a performance assessment of the proposed framework. Section 5 describes related work, and Section 6 concludes the paper.

2 Design and Implementation of Smart Proxies and Interceptors in RMI

It was important to build a model for smart proxies and interceptors that could be easily used, both in existing and in new applications. The objective of these abstractions is to simplify the architecture of the applications, so a complex model would defeat this purpose. It should be possible to deploy new service-specific functionality making minimal changes to the current application, either at the client side or at the server side. Ideally, only the server, that creates and registers the remote objects, should need to be changed. The clients should not need to be upgraded, to avoid the scalability issues that would arise from that. Finally, it was also important to minimize the overhead introduced by this framework, so that it is possible to use it in real world applications.

The best solution seemed to be building the model as a simple library, that could be used directly by applications. The developer would only need to implement a few interfaces, and use factory objects to create the required structure, without having to worry about the internal details or having to change the application structure.

2.1 Description of the Model

To add smart stubs to RMI it is necessary to place a layer of indirection both at the client and at the server. When exporting an object, the RMI runtime sends a stub to the client, which will represent the server object, taking care of communication between the client and the server. This stub is generated at compile time, by the `rmic` compiler. A possible solution is to change this stub. This solution is transparent to the client, which is not aware that it is receiving a modified stub. Unfortunately, this solution requires a separate compilation and debug step. For every remote object it is necessary to generate its stub, change it, and compile it again. If later on it is necessary to change the Remote object, the stubs must be regenerated and changed again to incorporate custom code. This destroys the independence between smart stubs and the application, making it an undesirable solution.

The solution that was used consists on creating the layer of indirection by adding more objects in the remote method invocation path between the client and the server. This is accomplished by using dynamic proxies, and by exploring the way that RMI handles different types of objects.

Our approach consists on replacing the stub by a dynamic proxy. When a client tries to bind with a Remote object, instead of the stub it receives a dynamic proxy, together with an invocation handler and other required objects, like interceptors. The invocation handler provides a hook which can be used to implement service-specific functionality. It must also forward the method invocation request to the server. For that, a standard RMI Remote object is used. This Remote object also provides a server side hook, that may be used to add interceptors. It has a reference to the real server object, and uses reflection to invoke the method. This structure is described in Fig. 1.

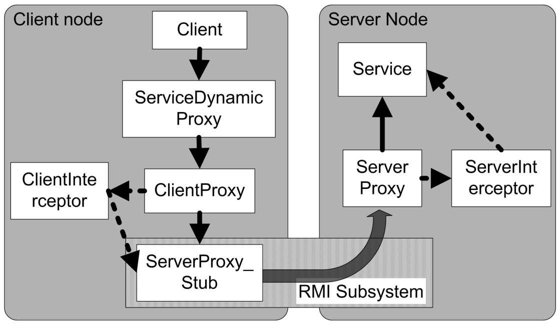

Figure 1: Smart proxies and interceptors in RMI

The **`Service`** object is the application's Remote object. It is not necessary to change it to use smart proxies and interceptors. **`Client`** repreents the client application. Like the `Service` object, it is not necessary to change it. **`ServiceDynamicProxy`** is the dynamic proxy of `Service` that the client receives instead of the standard stub. This proxy is generated at runtime and implements the interfaces exported by the remote object and the `Remote` interface, which allows the object to be placed in a registry. **`ClientProxy`** is the invocation handler of the dynamic proxy. It is a Serializable object, so it can be passed by value to the client, together with the `ServiceDynamicProxy`. **`ServerProxy/ServerProxy_Stub`** is the Remote object responsible for performing the communications between the client and the server. The `ClientProxy` will use this remote object to forward the invocations to the server. **`ClientInterceptor`** is an optional object, defined by the developer, that can be sent to the client. If one is present, the `ClientProxy` calls it before forwarding the method call to the server. **`ServerInterceptor`** is another optional developer-defined object, that remains at the server. If present, it is invoked by the `ServerProxy` when a remote method invocation is received.

When the server sends the `ServiceDynamicProxy` to the client, the RMI API passes it by value (it is Serializable). Together with this object, it will try to recursively send all objects that are referenced by the `ServiceDynamicProxy`. The `ServiceDynamicProxy` references the `ClientProxy` object, which is Serializable and so is passed by value. This object contains references to the `ServerProxy` and, eventually, to the `ClientInterceptor`. `ClientInterceptor` is Serializable and is passed by value. `ServerProxy` is a remote object so it is passed by reference, being replaced its stub. If the `ClientInterceptor` does not reference any other objects the process stops here.

This framework was implemented as a library. To create interceptors, a developer needs only to provide implementations of the `ServerInterceptor` and `ClientInterceptor` interfaces. Factory objects are available that will create the required structure, and return the dynamic proxy to be sent to the client. To create smart proxies it is necessary to change the default implementation of the `ServerProxy` or the `ClientProxy` implementations provided by the library, which can be done by deriving from those classes.

This discussion presented a general description of the model, but there are still some important details and technical issues that need to be mentioned. This is done in the next subsection.

2.2 Finer Details

2.2.1 Method marshalling

Another problem that must be addressed is marshalling the method invocation request. When using RMI directly, the stub of the remote object handles this task. That is lost when the service object stub is replaced by a dynamic proxy. Therefore, it is necessary to implement a mechanism for forwarding method calls to the server.

The dynamic proxy converts the method invocation in two objects that are sent to the invocation handler: a `Method` instance describing the method that was invoked, and an object array with the arguments passed by the client. The arguments, which must be Serializable or Remote (otherwise the RMI call would fail anyway), can be sent to the server directly. But the `Method` object is not Serializable and therefore cannot be sent directly, without first being encoded in a Serializable form.

To encode the `Method` object it is used a similar approach to RMI, where each method is assigned an unique number known both by the client and the server. In RMI, this numbering is generated at compile time by the rmic compiler, and sent to the client hard-coded in the stub of the application object that is being exported. The number is an hash of the name and the descriptor of the method [1, 6]. This solution is not possible on our framework, as the application objects to be exported do not have stubs, only a dynamic proxy that is generated at runtime. The solution found is to generate the method numbering at runtime, during object binding, both at the client and the server. To produce the same results on both sides, the lexicographic order of the method signatures is used.

2.2.2 Exception Handling

There are two type of exceptions that may be issued in a remote invocation: Remote exceptions, which represent problems with the communication infrastructure, and application exceptions, which are issued by application-specific code. In our framework, the second type of exceptions is propagated back to the client, just like RMI does.

Remote exceptions are not handled the same way as RMI does. The client is not required to deal with remote exceptions in a explicit way, nor the service is required to declare that it throws remote exceptions. This is possible because the service object is never exported, and so does not need to implement the `Remote` interface and to declare `RemoteException` in the throws clause of the methods of the exported interfaces. This way, the client does not need to handle `RemoteException`s if the service implements an interface that does not declare them.

When a `RemoteException` is raised, it may be handled in three different ways. First, if can be forwarded to an `ExceptionHandler`, a client side object for intercepting and handling all remote exceptions received by the client. This object is created at the server and sent to the client associated with the `ClientProxy`. Second, the exception can be sent to the client application as a `RemoteException`, just like the RMI runtime does. This happens if the remote interface exported through the dynamic proxy extends `Remote`, and no `ExceptionHandler` is present. Finally, it can be sent to the client wrapped in a `RuntimeException`. This happens if there is no `ExceptionHandler` and the remote interfaces do not extend `Remote`.

It is up to the application developer to decide which semantic to use for remote exceptions. Having an `ExceptionHandler` at the client is useful in some situations, as it allows the developer responsible for the server to define the policy to handle communication failures, and to implement it on a central place at the client side. This releases the client application from the burden of handling communication failures. Hiding remote exceptions from the client is useful in some situations. For instance, when transforming stand-alone in distributed applications it is not necessary to change the client so that it catches remote exceptions, or to change the server objects to throw remote exceptions.

3 Practical Example: User Authentication and Authorization in RMI

The JAAS API allows multi-user applications to be built around the notions of Principals, Subjects and Permissions. A *Principal* is an identity, like a name, a Social Security Number, or any type of id. A *Subject* is a collection of principals that belong to the same entity, be it a person, an institution or another abstraction. *Permissions* are associated with principals in the JAAS policy files. When a thread is executing, it is associated with a certain Subject. Access control decisions are made based on the the permissions associated with the current Subject in the policy files.

The problem with RMI is that it does not extend this model to remote calls. When a client thread, running on behalf of a certain Subject, makes a remote call to a server, the identity of the Subject is lost. It does not matter if the code at the client is executing on behalf of a guest or an administrator. The server executes remote calls made by both users in the same security context. Another way of seeing this problem, is that when a server exports an object it was no way of granting different access privileges to different remote users.

To overcome the limitations of RMI, it was necessary to implement a design where remote method calls are executed under the identity of a certain user. For that, it is necessary to authenticate the user before executing the call, and then to implement access-control based on the authenticated user, by using a local policy.

In RMI, when a a Remote object is exported, clients can access it directly, bypassing any form of authentication. Therefore, to restrict access to a Remote object, it can not be exported directly. Instead, a `Login` object is exported, that clients will use to authenticate themselves. On successful authentication, the `Login` object creates, configures, and returns to the client a proxy for the service object. This proxy knows the identity of the user that was authenticated. When it receives a request from the client, it uses the security context of the authenticated user to execute the method. Figure 2 illustrates the login process.

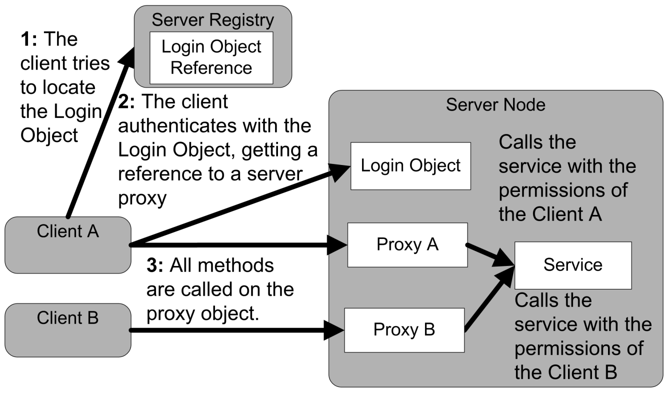
Server Registry
Login Object Reference
1: The client tries to locate the Login Object
2: The client authenticates with the Login Object, getting a reference to a server proxy
Server Node
Login Object
Calls the service with the permissions of the Client A
Client A
Proxy A
Service
3: All methods are called on the proxy object.
Client B
Proxy B
Calls the service with the permissions of the Client B

Figure 2: The authentication process for a Secure RMI server.

For each client a new proxy is created. So there is a one-to-one mapping between clients and proxies, although different proxies may be using the same service object.

After binding, when the client invokes a method in the proxy, the invocation is sent directly to the server. There is no need for an interceptor at the client, although in more complex systems a client side interceptors could be used to perform some security operation on all method invocations, like encryption or signing. At the server side, the method invocation is forwarded to a `SecureServerInterceptor`, whose function is to execute the invocation under the privileges of the user that was authenticated. This is accomplished using the `Subject.doAs()` method of the JAAS API, which can be used to execute a certain action under the security context of a given Subject. The `SecureServerInterceptor` invokes this method using the user's Subject. The action that is executed will simply call the target method, executing it inside the client's security context.

There are some issues that still need to be addressed for this solution to be secure, but further discussion is beyond the scope of this paper.

4 Performance evaluation

Adding layers of indirection to a RMI call degrades performance. A remote call using proxies and security will have the normal RMI overhead (RMI is still used to support the process), and the overhead associated with the extra object layers that are added, as well as the security operations

	Direct	Proxy			
		Proxy		Security	
Binding	5,418	6,402	18%	8,036	48%
void ping(void)	0,697	0,780	12%	0,898	29%
int ping(int)	0,716	2,082	191%	2,196	207%
SingleInt	1,927	1,968	2%	2,077	8%
MultiInts	2,177	2,217	2%	2,330	7%
StringTree	2,895	2,946	2%	3,056	6%

Table 1: Times for remote method calls. Values are in milliseconds and represent the average time it takes to complete a remote call. At the side of the values for the configuration with proxies is the percentage of overhead compared to direct RMI.

that must be done.

To better understand the penalty created by the framework proposed in this paper, a performance assessment was done. Three configurations were used: with conventional RMI (`Direct`), with a simple proxy without security (`Proxy`), and with security implemented using proxies, as defined in Sect. 3 (`Security`). When using a simple proxy interceptors are not used. The configuration with security has a server side interceptor.

In each configuration two types of tests were performed. One of them measures the time it takes for the client to bind with the server object. The other type of tests measures the time it takes to perform several different types of remote method invocations, depending on the type of arguments: none, primitives and objects. The invocations using objects as arguments were performed with three different types of objects: `SingleInt`, which contains only a field of the type `int`, `MultiInts`, which contains 20 fields of the type `int`, and `StringTree`, which is a `TreeSet` containing 32 strings with an average length of 12 characters.

The computers used in the test were two Pentium III at 800Mhz, with 512M of RAM running Windows 2000 with Service Pack 2. They were in the same LAN, connected by a shared 100Mbps Fast Ethernet. The java virtual machine we used was Sun's JRE 1.3.1_02. 10 series of tests were performed, each one consisting of 10.000 remote calls. The time for each remote method call was calculated based on the average of the results of each series. Table 4 shows the results.

The overhead of object binding is less than 18% when using proxies without security, but jumps to almost 50% overhead when using security. This overhead is caused by the need to create and send to the client more objects than when using direct RMI and, when security is used, by the authentication process. But this is a one time operation and, therefore, the extra time is not very important.

With direct RMI, the calls whose arguments do not contain objects have similar times: about 0.7ms. The time jumps to 2ms and more when using objects. The main difference between the two types of calls is that when sending objects as arguments, Serialization is used to marshal the objects. These results confirm the work of other authors [7] about Java Serialization. In its most generic form, object serialization is very slow. This is also noticeable on the tests with proxies, when using using primitives as arguments. The time jumps from 0.7ms with direct RMI, to 2ms with proxies. With direct RMI the primitive types are sent directly to the server. With proxies, they are wrapped on the corresponding wrappers class by the dynamic proxy, which causes the penalty of using serialization. The method `ping(void)` confirms these results, as it does not suffer the same overhead as the `ping(int)` method.

When using objects as arguments, the test with proxies has an overhead between 2%. In this situation, objects are sent both with direct RMI and with the proxies. The difference is that the proxy sends one more object, the object array that contains the arguments. It seems that there is a fixed overhead that RMI suffers when objects are sent, which does not scale when the number of objects per method call increases. After the first object, the overhead grows more slowly.

When using security, there is a fixed overhead of about 0.1ms for each method call, comparing to when using the same type of proxy with no security. This seems to be an acceptable overhead for most applications. This overhead is created at the server and derives from changing the security context under which the call is executed.

5 Related Work

For CORBA [8] there is the Portable Interceptor Specification [9], which defines the notion of interceptors. There is no standard specification for smart proxies in CORBA, but some ORBs have proprietary extensions to support them, like Iona OrbixWeb Orbix [10].

Wang [3] compares several approaches to meta-programming mechanisms for ORBs. It focuses on smart proxies and interceptors, and discusses the implementation issues related to them.

Other authors use smart proxies in CORBA to implement QoS support. Koster [2] describes how smart proxies can be implemented in CORBA. In his work, smart proxies and interceptors are used to provide QoS support. The advantage of using them is to separate the QoS protocol from the application specific code.

Microsoft's .NET framework [11] contains the Remoting Framework, which is a middleware for building distributed applications. It supports notions similar to smart proxies and interceptors, using the classes `RealProxy` and `TransparentProxy`, respectively.

Jini [12] defines a model to build distributed applications where it is possible to use something similar to smart proxies. Jini services are simply serializable objects that implement a specific interface. Clients search for services based on the interface, and receive a copy of them when they request it. The service may be a smart proxy that uses RMI or some other communication protocol in a transparent way to the client application. The disadvantages are that the programmer needs to build the smart proxy from scratch, and that Jini is a complex framework to learn and to deploy.

6 Conclusion

This paper argued that the default mechanism for generating and distributing stubs used in `RMI` lacks the flexibility required for some applications. There are several areas where this lack of flexibility manifests itself and that could benefit from the presence of smart proxies and interceptors in `RMI`. One of them is the difficulty in integrating security with `RMI`, as there are no mechanisms for implementing and enforcing user authentication and authorization at the level of the applications in remote calls.

In the first part of the paper, the notions of smart proxies (a stub that may perform more tasks than simple communication) and interceptors (objects that are called in the path of a remote invocation) were described. We believe these features should be part of `RMI`. We also demonstrated how it is possible to build a framework to implement those notions with total transparency for the client and for the application. All the work must be done at the server side, without changing application code.

In the second part, we demonstrated how it is possible to apply the framework to implement server side security in `RMI`, by integrating `RMI` calls with the Java Authentication and Authorization API (`JAAS`). We described a simple way to authenticate the principals associated with the clients, and to execute the remote method invocation at the server with the permissions of the authenticated principals.

In the last part of the paper, performance results were presented. The results show that the overhead ranges from under 5% to more than 300%, depending on the type of arguments of the remote method. We believe that for some types of applications this is perfectly acceptable.

Acknowledgements

This investigation was partially supported by the Portuguese Research Agency FCT (M&M Project - reference POSI/33596/CHS1999 and Scholarship SFRH/BM/6787/2001), and by CISUC (R&D Unit 326/97).

References

[1] Sun Microsystems Inc., "Java Remote Method Invocation Specification, JDK 1.3." Available at: `http://java.sun.com/products/jdk/rmi/`.

[2] R. Koster and T. Kramp, "Loadable Smart Proxies and Native-Code Shipping for CORBA," in *Proceedings of the Third IFIP/GI International Conference on Trends towards a Universal Service Market (USM), Munich*, pp. 202–213, September 2000.

[3] N. Wang, K. Parameswaran, and D. Schmidt, "The Design and Performance of MetaProgramming Mechanism for Object Request Broker Middleware," in *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01), San Antonio TX, USA*,, January 2001.

[4] Sun Microsystems Inc., "Java Dynamic Proxy Classes." Available at: `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`.

[5] Sun Microsystems Inc., "Java Authentication and Authorization Service JAAS." Available at: `http://java.sun.com/products/jdk/rmi/`.

[6] Sun Microsystems Inc., "The Java Virtual Machine Specification, Second Edition." Available at: `http://java.sun.com/docs/books/vmspec/`.

[7] M. Philippsen and B. Haumacher, "More Efficient Object Serialization," in *IPPS/SPDP Workshops*, pp. 718–732, 1999.

[8] Object Management Group, "The Common Object Request Broker: Architecture and Specification, 2.6," Tech. Rep. formal/2001-12-01, OMG, 2001.

[9] Object Management Group, "Portable Interceptors Specification," Tech. Rep. ptc/01-03-04, OMG, 2001.

[10] IONA Technologies, "Orbix 2000." Available at: `http://www.iona.com/products/orbix2000_home.htm`.

[11] Microsoft Corporation, ".NET Framework Developer's Guide." Available at: `http://msdn.microsoft.com/library/`.

[12] J. Waldo, "The Jini architecture for network-centric computing," *Communications of the ACM*, vol. 42, no. 7, pp. 76–82, 1999.