

Smart Proxies in Java RMI with Dynamic Aspect-Oriented Programming

Andrew Stevenson and Steve MacDonald
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada
{aastevenson, stevem}@uwaterloo.ca

Abstract

Java RMI extends Java with distributed objects whose methods can be called from remote clients. This abstraction is supported using statically-generated proxy objects on the client to hide network communication. One limitation of Java RMI is that these proxies can only forward method calls from client to server. Many applications can benefit from smart proxies that shift some application responsibilities to the client. Normally, developers must manually implement such proxies. This paper describes an aspect-oriented approach to creating smart proxies. This approach allows a server to extend an existing proxy with new capabilities at runtime. This approach is demonstrated with two examples.

1 Introduction

Java RMI (Remote Method Invocation) adds distributed objects to Java programs. Distributed objects live on *object servers* that run on separate computers connected by a network. Clients call methods on these remote objects using *remote method invocation*. The client assembles a message containing the information needed for the method call and sends the request to the server on which the receiver resides. The server executes the method and sends a reply message with the results.

An important characteristic of this remote object abstraction is that the developer is largely unaware of the supporting infrastructure. Most programming effort in Java RMI is directed at solving the problem at hand rather than the distributed environment in which the application executes. At the server side, the developer need only provide an implementation of the methods for the remote object and an interface describing the methods available to clients. At the client side, the client obtains a reference to the remote object and calls remote methods using that reference. This remote reference is hidden by a *proxy object* at the client, which acts as a local

placeholder for the remote object. This proxy forwards method calls to the server and processes replies, making remote calls appear as normal, local calls on the proxy.

In some applications, it is useful for proxies to take on additional responsibilities, producing *smart proxies*. Often, these responsibilities are optimizations to reduce the communication overhead of remote calls. Such responsibilities can include client-side caching to save remote calls, load balancing, and redistributing functionality from server to client to use computing power on the client. Further, smart proxies may need to change over time based on application needs and workload.

This paper presents a technique for adding new functionality to the default proxies produced for Java RMI using *dynamic distributed aspect-oriented programming* with the Java Aspect Components (JAC) system [12]. Aspect-oriented programming (AOP) allows code called *advice* to be inserted at specific points in the execution of a program called *join points*. *Dynamic AOP languages* allow advice to be added or removed at runtime. JAC adds the ability to distribute advice across a network. We use JAC to apply advice to the proxy to add new functionality. This ability allows us to construct a smart proxy from the simple proxies generated for Java RMI. To illustrate this approach, we present two examples, client-side caching and client-side input validation.

The research contributions of this paper are as follows. First, we use a dynamic, distributed AOP system to modify code on the client side. In contrast, most work in this area focuses on allowing the client to supply aspects to modify the server object implementation. Second, we show the ability to leverage the default proxies for Java RMI and make smart proxies from them at runtime. Using aspects, we do not require any changes to the existing Java RMI application development cycle or toolset. Last, we can leverage the dynamic characteristics of JAC to change the smart proxies during execution without having to redistribute and reinstantiate proxies at the client. Instead, JAC can distribute the new advice and apply it to existing proxy objects that have already been instantiated on clients.

2 Background Information

Architecture of Java RMI Java RMI is based on the distinction between the remote interface that defines the methods available to clients and the object implementation that provides the behaviour. Two objects implementing the same interface are indistinguishable if they provide identical behaviour.

The architecture of Java RMI consists of the three layers in Figure 1 [2]. The first layer provides a proxy object at the client. Older versions of Java RMI (Java 1.1 and earlier) generated a skeleton object at the server for each remote class, but Java Reflection obviated the need for class-specific skeletons. The proxy object is an local object on the client that implements the same remote interface exported by the server object. The proxy transforms local method calls by the client into remote calls to the server object. Part of this translation requires information about the *remote object reference* for the remote object. This information is held in the Remote Reference Layer. Finally, the Transport Layer is responsible for network communication between client and server.

In Java RMI, the client-side proxy can be automatically constructed in one of two ways. The first way is to statically generate the proxy during application development using the `rmic` stub compiler included with Java. Given a class that implements an interface derived from the interface `java.rmi.Remote`, the `rmic` compiler generates a proxy class that implements the same interface. The name of the proxy class is the name of the implementation with “_Stub” appended. For each method in the interface, the stub compiler produces code that uses the remote object reference to invoke the same method on the server implementation. At runtime, when the client imports the remote object using the RMI registry, it attempts to load the proxy class based on its name. If the proxy class is successfully loaded, the proxy is created. If the proxy class cannot be found, then the second method of proxy creation is used.

The second method of constructing a client-side proxy is the *dynamic proxy* mechanism introduced in Java 1.3 [14]. Given a list of interfaces, the JVM can create a proxy object implementing all of them at runtime. Methods in the dynamic proxy are delegated to an *invocation handler* provided by the user. In Java RMI, if the proxy class could not be loaded by the client (as described above), a dynamic proxy is created using the remote interface of the server object. A `RemoteObjectInvocationHandler` is created as the invocation handler, which has the same functionality as `rmic`-generated stubs. We consider these dynamic proxies to be statically-generated since their functionality is fixed; they are dynamic only in that they are created at runtime without a separate code generation step.

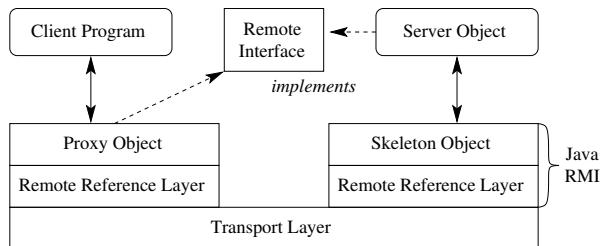


Figure 1. Java RMI Architecture (from [2]).

AOP in JAC Aspect-oriented programming was created to deal with *cross-cutting concerns* [4]. A cross-cutting concern is functionality that cannot be encapsulated into one place using classes and methods. Instead, this functionality must be spread throughout the system, resulting in tangled code that can be difficult to maintain.

An example of such a concern is method logging, where the entry and exit of all public methods should be recorded. This functionality cannot be isolated in a single method or class, but must be inserted at the start and end of every logged method. This introduces several development and maintenance problems. Developers must be aware of this requirement and be sure to include logging in new methods, and add it to existing methods that are made public. If the interface to the logging class changes, the code in all public methods may be affected.

AOP is based on two main ideas: *join points* and *advice*. A join point is a well-defined event in the execution of a program, like a method call, method return, or field access. A *pointcut* is a set of join points of interest. Advice is code that runs when an event in the pointcut is matched. In general, advice can run before, after, or around the event. In the latter case, a special method is called in the advice to execute the code associated with the event. Pointcuts and advice are encapsulated into an entity called an *aspect*. The aspect and application source code are combined in a process called *weaving*, which takes at compilation time or at run time. The latter produces a *dynamic aspect-oriented system* where aspects can be applied while an application runs.

Because join points are language-specific, different languages have different AOP tools. For Java, one tool is Java Aspect Components (JAC) [12], a dynamic aspect-oriented software development kit that allows aspects to be woven at runtime and distributed over a network. Unlike other Java-based AOP languages, JAC uses classes and methods rather than new language constructs.

In JAC, a pointcut is built with the following method:

```
void pointcut(String objects, String classes, String methods, Wrapper advice);
```

The first three arguments are modified regular expressions with extra syntax for describing pointcut-related abstractions. The `objects` argument specifies the objects in the pointcut. JAC allows individual objects to

```

1 public class LoggingAC extends AspectComponent {
2     public LoggingAC() {
3         pointcut(".*", "ca.uwaterloo.*",
4             "login(String):boolean", new LogWrapper());
5     }
6 }
7 class LogWrapper extends Wrapper {
8     public Object invoke(MethodInvocation mi) {
9         Object[] args = mi.getMethod().getArguments();
10        Logger.add("Login: " + args[0].toString());
11    }
12    Boolean loginOK = (Boolean) proceed(mi);
13
14    if (loginOK)
15        Logger.add("Successful login");
16    else
17        Logger.add("Failed login");
18    return (loginOK);
19 }
20 }

```

Figure 2. A logging aspect written in JAC.

be advised, based on its own object identifiers. The `classes` argument specifies the classes that are part of the pointcut. The `methods` argument specifies the methods in the identified classes and objects that are to be advised. The syntax is “`methodName(-argType1, ...):returnType`.” The `advice` argument is a reference to a JAC wrapper object with advice to be woven into the join points for the pointcut.

An example logging aspect in JAC is given in Figure 2. The pointcut on line 3 captures calls to the method `login(String)` in objects of classes in the package `ca.uwaterloo`. When these events occur, the advice in a JAC wrapper of type `LoggingWrapper` runs, given at line 7. The `invoke()` method on line 8 gives the advice. This method has a parameter of type `MethodInvocation` that gives access to information about the join point that was triggered, such as method name and arguments. JAC only supports around advice, so the method for the join point must be explicitly invoked using the `proceed()` method in line 12. Note that advice can alter arguments and return values, or bypass the advised method by not calling `proceed()`.

Aspects in JAC have two other important properties that we exploit in this work. First, JAC is a dynamic aspect system. New aspects can be created at runtime by creating new pointcuts and associated JAC wrappers, and existing aspects can be removed from a running system. Second, JAC can add and remove aspects on remote machines. This is done in one of two ways. First, the pointcut specification can be extended with a *host pointcut expression*, an extra argument to `pointcut()` that specifies the hosts to which the pointcut applies. This argument could be provided using constructor arguments so it can be dynamically applied to any host. Second, the `remoteWeaveAspect()` method can be used to forward an aspect to a JVM running on remote host and weave the advice into the running application.

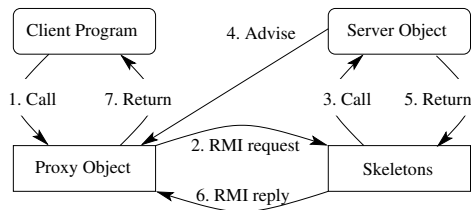


Figure 3. Advising proxies with aspects.

JAC has several features and aspects that are not used in this work. JAC has a persistence aspect to load and store objects to disk, a transaction aspect that rolls back object changes on error, and a synchronization aspect.

3 Advising Java RMI Proxies

The basic outline of our approach is shown in Figure 3. In this approach, the server object applies advice to the client-side proxy during a remote method call. To synchronize the application of the aspect by the server with the execution of the proxy on the client, advice is applied while the client is blocked waiting for a reply to a remote method. The advice is woven into the client code and affects subsequent calls made by that client.

Advice is sent from server to client using the method `remoteWeaveAspect()`. The client host is obtained using the `RemoteServer.getClientHost()` method. When advice reaches the client, it is woven into the proxy and will be executed on later calls.

This idea is demonstrated with a caching example. The server code is given in Figure 4 and the caching aspect in Figure 5. Lines 8 through 14 in Figure 4 show the server applying advice to the client. For simplicity, this code is in the server methods, though it may also be a cross-cutting concern and implemented using another aspect. Note that the advice is applied while the client is blocked waiting for its call to `factorial()` to complete, which removes many concurrency issues. `remoteWeaveAspect()` applies the advice in the class `CachingAC` to the stub on the client host, as noted in the pointcut starting at line 7 in Figure 5. Client source code is unaffected by the application of advice; example client code is given in Figure 6.

This solution is simple but has a serious drawback: the list of hosts to which the aspect has been applied can be out of date. This can happen when a client terminates since the server is not notified of this event. The advised code is lost since JAC weaves bytecode in memory, so later clients on the same host load an unadvised version of the proxy. A more complex solution, that does not alter the RMI protocol or client, is to apply the aspect when the frequency of remote calls exceeds a threshold.

We assessed the overhead of advising a method with

a microbenchmark. A remote object was created with a method with no arguments, return value, or method body. The time to call this method was 0.495 ms. The server applied an aspect that only called `proceed()`, and the time increased by 2.6%, to 0.508 ms.

This approach has several benefits from the perspective of the application developer. The first and largest benefit to this aspect-oriented approach is that it does not introduce any new tools or extra steps into the development process. Programs are built with the standard Java compiler and proxies are generated with the standard `rmic` stub compiler. This reduces the learning curve for applying this technique to applications.

Second, since the tools used to develop applications are unchanged, this technique is easily applied to existing applications. The server-side implementation must still be changed to create and distribute aspects to the client. However, it is not necessary to change any other components in a distributed object program.

Third, JAC allows aspects to be added and removed at runtime. Aspects for load balancing and caching can be applied by a server under heavy load and removed when the load falls. The data replacement scheme for a cache can be changed if consistency requirements change.

This approach has limitations that prevent an application from being oblivious to the use of aspects to modify proxies. First, the client must be run in a JAC-aware container so remote aspects can be applied. The client and server must be started using “`java -jar jac.jar configFile.jac`” where `configFile.jac` indicates the application name and the class with the main method. Note that the client code does not contain any references to the JAC configuration or JAC library code; only the startup procedure is different. The server includes JAC library calls to create and distribute advice.

Second, the client proxy has no mechanism for making its JAC identifier available to the server. As a result, server aspects target all instances of the proxy class on the client. In the future, we would like to exploit the per-object advice feature in JAC to remove this limitation.

Third, the server implementation must create any aspects that may be sent to clients. This requires design effort to factor these concerns into aspects. However, similar effort is needed in any implementation of smart proxies that allow functionality to change at runtime.

Last, introducing aspects with JAC adds overhead. Aspects must be sent from server to client, adding messaging overhead. This could be reduced by piggybacking the aspect with the results of a remote call, but this would require changes to the RMI protocol, making this approach more intrusive. These aspects must then be woven into the client code. The expected benefits of the applied aspect must outweigh these costs.

```

1 public class Calculator implements MathOps
2 {
3     private List<String> clients =
4         new ArrayList<String>();
5
6     public int factorial(int n)
7         throws RemoteException {
8         // Apply advice to new clients only.
9         String host = RemoteServer.getClientHost();
10        if (!clients.contains(host)) {
11            Jac.remoteWeaveAspect("TestApp", host,
12                "CachingAC", "config.acc");
13            clients.add(host);
14        }
15        if (n == 0) return 1;
16        int result = 1;
17        for (int i = 1; i <= n; i++)
18            result *= i;
19        return result;
20    }
21 }

```

Figure 4. Applying caching aspect to clients.

```

1 public class CachingAC extends AspectComponent
2 {
3     private Hashtable<Integer,Integer> cache =
4         new Hashtable<Integer, Integer>();
5
6     public CachingAC() {
7         pointcut(".*", "Calculator_Stub",
8             "factorial(int):int",
9             new CachingWrapper());
10    }
11
12    public class CachingWrapper extends Wrapper
13    {
14        public Object invoke(MethodInvocation mi)
15            throws Throwable {
16            Integer arg = (Integer) mi.getArguments()[0];
17            Integer result = cache.get(arg);
18            if (result == null) {
19                result = (Integer) proceed(mi);
20                cache.put(arg, result);
21            }
22            return result;
23        }
24    }
25 }

```

Figure 5. The caching aspect applied to a client in Figure 4.

```

1 public class Student
2 {
3     public static void main(String[] args) {
4         try {
5             MathOps c = (MathOps) Naming.lookup(args[0]);
6             for(int i = 1; i < args.length; ++i) {
7                 int num = Integer.parseInt(args[i]);
8                 int answer = c.factorial(num);
9                 System.out.println(num + "! is " + answer);
10            }
11        } catch (Exception e) {
12            e.printStackTrace();
13        }
14    }
15 }

```

Figure 6. Client code for example program.

3.1 Comparison to Other Approaches

In this section, we compare our AOP approach to smart proxies with other methods. We first describe approaches based on the methods of generating proxies in Section 2. The last section describes other approaches.

Rewriting `rmi`-Generated Proxies The first approach to producing a smart proxy is to manually create or subclass a proxy object for the client [8]. To serve as a proxy, a class have the following characteristics:

- It extends `java.rmi.server.RemoteStub`.
- It implements the remote interface supplied by the developer and the `java.rmi.Remote` interface.
- It is named `<Implementation>_Stub`, where `Implementation` is the name of the class with the server implementation of the remote interface.

Any class with these characteristics is remotely loaded by the client, and instances used as proxies. A simple way to meet these characteristics is to use the `-keep` option with `rmi` to keep the source code for the generated proxy. This class can be converted to a smart proxy.

The main benefit to this approach is that it provides complete control to the developer to create a smart proxy. However, altering the functionality of the smart proxy at runtime is difficult without extending the proxy to include an extra Strategy object [1] or similar mechanism. The RMI protocol will also need to be extended to pass this object between server and client. Furthermore, it increases the costs of development and maintenance as both server implementation and proxy must be maintained, where our aspect-oriented approach puts the development effort in the server implementation only.

Dynamic Proxies The second approach is to use the dynamic proxy facility in Java. All calls to the dynamic proxy are dispatched to an invocation handler associated with the proxy. This handler can use the RMI protocol to forward method invocations to the server. This approach has been used in distributed middleware research like Babylon [15] and RMIX [5]. In Java 1.5, this approach is used when `rmi`-generated stubs are not found.

Like altered `rmi`-generated proxies, the developer has full control over the behaviour of the proxy. However, changing its behaviour at runtime is still difficult. Further, Java RMI cannot create custom invocation handlers for new dynamic proxies. Instead, the remote object registry or its lookup methods must be altered to create the desired proxy. RMIX provides its own version of the `Naming` class to create its proxies from the remote reference returned by `rmiregistry`. Regardless, the use of dynamic proxies affects the client code.

Other Approaches FORMI is built on a *fragmented object model*, where an object implementation is split

across fragments that can be individually distributed across a network [3]. Fragments can be used to replicate or partition objects across the network. In FORMI, a proxy is a fragment on the client. A fragment can be replaced with another during execution, allowing the distribution of responsibilities to change at runtime. This is accomplished using a layer of indirection similar to the Strategy pattern [1]. FORMI combines the proxy object and remote reference layers in Java RMI by providing a replacement for `rmi` that produces *FORMI stubs*.

FORMI provides the same flexibility as our aspect-oriented approach, and also does not impact client code. However, FORMI introduces its own stub compiler. Furthermore, the implementation of a remote object must be explicitly divided into fragments, each of which must subclass a FORMI library class. This can impact the ability of the user to reuse existing code in the fragments.

Santos *et al.* solve this problem by constructing a general framework to introduce smart proxies and *interceptors*, extra objects that are introduced in the flow of control in both client and server for remote objects [13]. This framework exploits the dynamic proxy mechanism in Java. It is not clear from the available papers and resources how smart proxies are introduced and configured, and if they can be changed at runtime.

4 Client-Side Input Validation

This section presents an another example of advising proxies with JAC using client-side input validation. To keep the discussion simple and focus on the aspects rather than the application, the example uses the factorial method used earlier. Client code that calls this code was presented in Figure 6. The client code does not change to accommodate the applied aspects; only the server code changes, to apply aspects to the client.

In client-side input validation, the client can cut network overhead by locally evaluating preconditions that do not rely on server-side state. The return value or exception for these inputs can be generated without sending a message to the server. This is shown in the aspect given in Figure 7. If the argument is 0 or 1, then the value 1 is returned. The aspect could also check for illegal argument values and throw an exception. The server object is only invoked if the argument is a positive value. It may also be beneficial to apply a server-side aspect to circumvent the validation checks that have already been performed at the client. This is easiest if the checks are in methods that can be bypassed by an aspect.

One potential problem with the use of aspects is ordering multiple pieces of advice applied to the same join point. For instance, if caching and validation aspects are applied to the same method, the order in which they run

```

1 public class ValidationAC extends AspectComponent {
2     public ValidationAC() {
3         pointcut(".*", "Calculator_Stub",
4             "factorial(int):int", new ValidateWrapper());
5     }
6     public class ValidateWrapper extends Wrapper {
7         public Object invoke(MethodInvocation mi)
8             throws Throwable {
9             Integer arg = (Integer) mi.getArguments()[0];
10            if (arg == 0 || arg == 1)
11                return(1);
12            return proceed(mi);
13        }
14    }
15 }

```

Figure 7. Example validation aspect.

may be important so the cache does not have entries for calls with invalid arguments that throw exceptions. JAC provides a mechanism for specifying advice order.

Security concerns are another issue in this approach. We assume that clients trust the server to advise proxies. However, our approach has a basic security mechanism in that the client must explicitly run the proxy in a JAC-aware container. Other research has considered finer-grained security mechanisms [6, 11].

5 Other Related Work

Other aspect-oriented systems have attempted to address distributed object concerns. Some research tools use AOP to solve problems like caching, persistence, and replication. More general distributed aspect-oriented systems have also been developed.

The JAC framework includes aspects for persistence, transaction semantics, synchronization, caching, process monitoring, replication consistency, and load balancing. It also supports the distribution of both pointcuts and advice, which enables us to create smart proxies.

The remote pointcuts in DJcutter allow developers to write pointcuts that specify not just a specific event in the execution but also the host on which the event takes place [10]. These pointcuts are distributed to the object servers in the application. However, all advice is run on a designated *aspect server* rather than on the host that triggered the pointcut, making it unsuitable for smart proxies. D supports distribution in its aspects, but also cannot distribute the execution of advice [7].

AWED provides a more flexible model, distributing pointcuts and advice separately [9]. A pointcut can specify the hosts on which the join point applies, and the hosts on which advice for the pointcut should be run. Note that advice for a given pointcut may run on several hosts, which can be used to implement consistency mechanisms for replicated data. This research could have used AWED in its implementation. One weakness of AWED is that it uses new language constructs to spec-

ify pointcuts that cannot be parameterized with runtime arguments, unlike the `pointcut()` method in JAC.

6 Conclusions

In this paper, we presented an aspect-oriented approach to constructing smart proxies in Java RMI. These smart proxies augment `rmi`-generated proxies with additional application functionality. The primary benefits to our approach are that it does not alter the existing development or deployment tools, that it does not impact client code so existing RMI code can be used, and that the functionality in the smart proxy can be added and removed at runtime. This approach was demonstrated using two examples. While the use of this technique is not completely oblivious to client programs, we believe the technique is still useful for adding and removing smart proxies in a distributed object system.

References

- [1] E. Gamma, *et al.* *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [2] jGuru. *Remote Method Invocation*, 2000. <http://java.sun.com/developer/onlineTraining/rmi/RMI.html>.
- [3] R. Kapitza, *et al.* Formi: Integrating adaptive fragmented objects into java rmi. *IEEE Distributed Systems Online*, 7(10), 2006.
- [4] G. Kiczales, *et al.* Aspect-oriented programming. In *Proc. 11th European Conf. on Object-Oriented Programming, LNCS* vol. 1241, pages 220-242, 1997.
- [5] D. Kurzyniec and V. Sunderam. Semantic aspects of asynchronous RMI: The RMIX approach. In *Proc. 6th Intl. Workshop on Java for Parallel and Distributed Computing*, 2004.
- [6] D. Larochelle, *et al.* Join point encapsulation. In *Proc. 2003 Workshop on Software Engineering Properties of Languages for Aspect Technologies*, 2003.
- [7] C. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.
- [8] T. Loton. The smart approach to distributed performance monitoring with Java. <http://www.javaworld.com/javaworld/jw-09-2000/jw-0901-smart.html>, Sept. 2000.
- [9] L. Navarro, *et al.* Explicitly distributed AOP using AWED. In *Proc. 5th International Conf. on Aspect-Oriented Software Development*, pages 51-62, 2006.
- [10] M. Nishizawa, *et al.* Remote pointcut: A language construct for distributed AOP. In *Proc. 3rd Intl. Conf. Aspect-Oriented Software Development*, pgs 7-15, 2004.
- [11] H. Ossher. Confirmed join points. In *Proc. 2006 Workshop on Software Engineering Properties of Languages for Aspect Technologies*, 2006.
- [12] R. Pawlak, *et al.* JAC: An aspect-based distributed dynamic framework. *Software: Practice and Experience*, 34(12):1119-1148, 2004.
- [13] N. Santos, *et al.* A framework for smart proxies and interceptors in RMI. In *Proc. 15th ISCA International Conf. on Parallel and Distributed Computing Systems*, 2002.
- [14] Sun Microsystems, Inc. *Dynamic Proxy Classes*, 2004. <http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html>.
- [15] W. van Heiningen, *et al.* Exploiting dynamic proxies in middleware for distributed, parallel, and mobile java applications. In *Proc. 8th International Workshop on Java for Parallel and Distributed Computing*, 2006.