

# Dynamic Aspect-Oriented Load Balancing in Java RMI

Andrew Stevenson and Steve MacDonald

David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, CANADA  
{aastevenson, stevem}@uwaterloo.ca

## Abstract

*Load balancing is the process of distributing client requests over a set of servers, and is a key element of obtaining good performance in a distributed application. Java RMI extends Java with distributed objects whose methods can be called from remote clients. In some Java RMI programs, there may be multiple replicas of a given object that can be the receiver of a remote method invocation. Effectively distributing these requests across these replicas requires either an extra balancer process or additional code on the client for this distribution. In this paper, we demonstrate the use of dynamic aspects in JAC to solve this problem. The client proxy is modified with an aspect to forward requests to a specific server, but the server is also able to shed load by altering or removing this aspect. The overhead of this approach is evaluated using a set of microbenchmarks.*

*Keywords:* Load balancing, Java RMI, Dynamic aspect-oriented programming.

## 1 Introduction

Java RMI (Remote Method Invocation) adds remote objects to Java programs. These remote objects reside on *object servers*, separate machines connected by a common network. Clients can invoke methods on these remote objects using *remote method invocation*, which bundles the information needed to invoke the method into a message and sends it to the appropriate object server for execution.

In compute-intensive remote object programs, clients may be invoking many expensive methods on servers. In particular, they may be invoking expensive methods on the same object, running the risk of decreasing performance by overloading the server. To improve performance this object can be replicated on several servers and requests can be distributed to a suitable replica, normally the server with the lightest load. This distribution of requests is referred to as *load balancing*, and is key to good performance in many distributed applications.

For this paper, we are specifically interested in dynamic load balancing; we do not assume that workload is constant and predictable enough that statically allocating replicas to clients will produce an optimal solution. We also assume that while the client must participate in a load balancing scheme, it does not have sufficient information to determine the best strategy for invoking remote objects. That is, we assume the implementation of the remote objects is in the best position to select an appropriate load balancing strategy.

This paper describes an approach to load balancing in Java RMI based on *dynamic distributed aspect-oriented programming* using Java Aspect Components (JAC) [13]. Aspect-oriented programming (AOP) allows code (in the form of *advice*) to be inserted at specific points in the execution of a program. Dynamic AOP systems allow advice to be added and removed at runtime. JAC also allows advice to be transmitted over a network and applied to objects running on other object servers. We use these capabilities to create a balancer process that modifies a proxy object on the client to direct its remote method invocations to a specific replica. This advice can be altered or removed by a server to redistribute the invocations to other replicas and shed load.

The research contributions of this paper are as follows. First, it shows the use of a dynamic, distributed AOP system to modify proxy code on the client. In contrast, most work in this area focuses on allowing the client to supply aspects to modify the server object. Second, this load balancing strategy is controlled by the balancer and server processes based on their knowledge of application requirements. The strategy can be altered at runtime by these processes as necessary.

This paper is organized as follows. Section 2 presents background material covering several topics important to this research, including Java RMI, aspect-oriented programming with JAC, and load balancing. Section 3 describes our approach to load balancing with JAC. We measured the overheads in our approach using a set of microbenchmarks; these results are given in Section 4. Additional related research is presented in Section 5, and the paper concludes with Section 6.

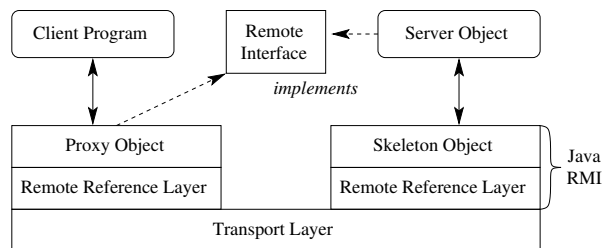


Figure 1. Java RMI Architecture (from [3]).

## 2 Background Material

### 2.1 Java RMI Architecture

Java RMI is based on the distinction between object interface and implementation. It relies on the fact that a client cannot distinguish between objects implementing a remote interface if their behaviour is identical.

The architecture of Java RMI consists of the three layers in Figure 1 [3]. The first layer provides a proxy object on the client and a skeleton object at the server. In current versions of Java, there is one skeleton object for the server. The proxy object is a local object on the client JVM that implements the same remote interface as the object implementation on the server. The proxy translates method invocations to remote method invocations to the server. Part of this translations uses the *remote object reference* for the remote object held in the Remote Reference Layer. The Transport Layer handles client/server communication.

The proxy object may be statically-generated by the `rmic` stub compiler or may be a *dynamic proxy* generated at runtime by the JVM. The `rmic` compiler starts with a class that implements a remote interface (one derived from `java.rmi.Remote`). From this, `rmic` generates a proxy class that implements the same remote interface. The name of this proxy class is the name of the implementation with “\_Stub” appended. For each method in the remote interface, `rmic` generates code that uses the remote object reference to invoke the same method on the object implementation at the server. At runtime, when the client imports the remote object using the RMI registry, it loads this proxy class using its name. If the proxy class is successfully loaded, a proxy object is created. If not, then the second method of proxy generation is used.

The second method of generating a proxy object is using the *dynamic proxy* mechanism introduced in Java 1.3 [16]. Given a list of interfaces, the JVM can create a proxy implementing them at runtime. Method calls on the proxy are delegated to an *invocation handler* object provided by the developer. In Java RMI, if the

JVM cannot load the `rmic`-generated proxy class, the client creates a dynamic proxy using the remote interface. A `RemoteObjectInvocationHandler` object is created as the invocation handler, which provides identical functionality as `rmic`-generated stubs. However, we consider these dynamic proxies to be statically-generated. Their functionality is fixed; they are dynamic only in that they are created at runtime.

### 2.2 Software Load Balancing

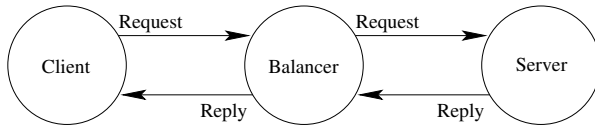
In this section, we’ll examine some software-based load balancing approaches that have been applied to Java RMI and similar distributed systems.

The first approach is perhaps the most obvious. A load balancer process is placed between clients and servers. All client requests are forwarded to the balancer process, which forwards the request to a suitable server. The reply message takes the reverse path. This is shown in Figure 2. In Java RMI, the balancer would maintain a collection of references to different remote objects. For each incoming request, one of these remote objects would be selected and the balancer would invoke the same method on it, forwarding the request. The return value of the balancer method would simply forward the return value from the remote object. A similar strategy can be used in Apache, forwarding all requests to an entry server that rewrites the URL to redirect the request to one of a set of servers [1].

This strategy has the benefit of being able to redirect each request from each client to a suitable server. In addition, incorporating new servers is relatively simple. When a new object starts on a new server, it could register itself with the balancer. From that point, the balancer could distribute requests to the new object. The balancer can also control the load on the servers by deciding how many requests to forward to any given server. Once this number has been reached, the balancer could queue up requests and forward them to servers as they complete their outstanding requests.

In addition, this strategy allows the servers (in conjunction with the balancer) to shed load when necessary. The balancer can factor in server load when distributing requests, by having each server periodically indicate its current status. The client is not involved in this process, instead simply forwarding all requests to the central balancer process.

However, this strategy adds communication overhead in the extra pair of messages between balancer and server. This overhead can be reduced by having the server reply directly to the client, which is not possible in Java RMI without altering the underlying RMI protocol. In addition, the balancer can potentially form a bot-



**Figure 2. A balancer forwarding requests and replies.**

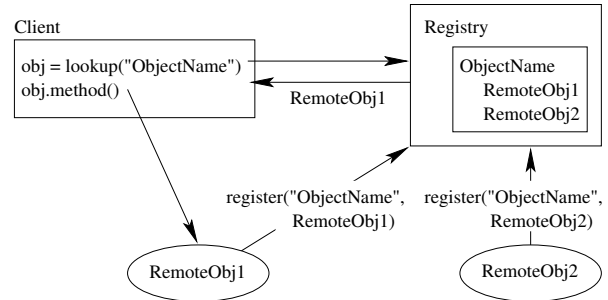
tleneck since all requests must pass through it, though the amount of processing for each request is small.

A second option is to augment the object registry to allow multiple remote objects to register remote object references using the same name. When a lookup is performed, the registry can return one of the registered references to the client, and the client invokes methods directly on that object. This approach is shown in Figure 3.

This approach cannot be implemented using the RMI registry supplied with Java RMI. That registry does not permit multiple entries for the same name, instead throwing an exception on successive attempts to bind with the same name. Instead, a customized registry must be used. Examples of this approach include the SmartRegistry [9] and Jgroup/ARM [8], though both systems allow multiple registrations to support replication rather than load balancing. In these systems, the registry returns a proxy with references to all available replicas for group communication. In a load balancing system, the registry would return a single reference to one of the registered objects.

Another example of this approach is DNS load balancing. DNS databases can have multiple IP addresses associated with a name, and DNS servers can be configured to return different addresses according to a policy. DNS load balancing has several limitations. First, it can only return different IP addresses, so all servers on these hosts must use the same port number. Second, DNS does not check the availability of a machine before returning its address, so it may return the address of a crashed server.

A problem with this strategy is that the client generally caches the object reference and uses it for all remote method invocations, which limits the ability to balance load. That is, rather than redirecting each request from each client to a different server, this strategy redirects all requests from a client to the same server. It is possible that different clients will issue a different number of remote calls and will cause different loads at their servers. To redirect requests to another server, a client would need to obtain a new remote reference by issuing another lookup to the registry, and ensure that all client code uses the new remote object. In DNS load balancing, this problem is exacerbated by the ability of intermediate name servers to also cache results, although this



**Figure 3. Balancing using multiple registry entries.**

can be mitigated by setting the time-to-live field on the address entry to expire relatively quickly.

Another serious problem with this strategy is that it falls to the client to detect and redistribute the load if the server becomes overwhelmed, by reissuing a lookup request. There is no simple mechanism for allowing the server to automatically reduce its load by not accepting new requests.

### 2.3 AOP in JAC

Aspect-oriented programming was created as a way to address *cross-cutting* concerns that appear in source code [5]. A cross-cutting concern is functionality that cannot be isolated in one single class or method, but rather is spread throughout the code in a system. The result is tangled and scattered code that is difficult to develop and maintain.

An example of a cross-cutting concern is method logging, where the entry and exit of all public methods should be recorded. This functionality cannot be isolated in a class or method in current object-oriented languages. Instead, logging code must be placed at the start and end of every relevant method. This introduces several development and maintenance problems. Developers must be aware of this requirement as they add new classes and methods to the application. Maintainers must be aware of this requirement if they make existing methods public. Finally, if the logging interface changes, it may affect the code in all public methods.

AOP addresses these problems using two main ideas: *join points* and *advice*. A join point is a well-defined event in the execution of a program. Events can include method calls, method returns, or field accesses. A *pointcut* is a subset of the join points in a program that are of interest to the developer. Advice is code associated with a pointcut. This code is run whenever a join point in its pointcut is encountered in the execution of a program. In general, advice may run before, after, or around the pointcut. In the latter, the advice must explicitly ex-

ecute the code matching the join point using a special method. Pointcuts and advice are often bundled into an entity called an *aspect*. Aspects and application source code are combined in a process called *weaving*, which may take place at compile time or run time. The latter produces a *dynamic aspect-oriented system* that allows aspects to be added and removed as the application runs.

Because join points are language-specific, different languages have different AOP tools. For Java, one tool is Java Aspect Components (JAC) [13]. JAC is a dynamic AOP system that allows aspects to be woven at runtime, and further allows aspects to be distributed over a network. Unlike many other Java-based AOP tools, JAC uses classes and methods to specify AOP constructs rather than language extensions.

In JAC, a pointcut is built using the method:

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

The first three arguments use a modified regular expression syntax to specify the object, classes, and methods for the pointcut. JAC allows individual objects to be advised using internal object identifiers. The last argument, *advice*, is a reference to a *JAC wrapper object* that holds advice to be woven into the join points specified for the pointcut. JAC only supports around advice for method invocations, so the code for the join point must be explicitly invoked using a special method called `proceed()`. The return value of the advice is used as the return value of the join point where applicable.

An example logging aspect in JAC is given in Figure 4. The pointcut on line 3 captures method calls to the method `boolean login(String)` in any object of any class within the package `ca.uwaterloo`. When these events occur, the advice in the `LogWrapper` class runs. This class is given on line 7. Specifically, the `invoke()` method at line 8 executes. This method takes an argument of type `MethodInvocation`, which provides information about the specific join point that triggered the execution of this advice, such as the name and arguments of the called method. The arguments are used to print a log message including the name of the user issuing the login attempt. The `login()` method is called using the `proceed()` call at line 12. The advice can alter the arguments to the method or its return value, or even bypass the advised method by not calling `proceed()`.

Aspects in JAC have two important properties that we will exploit in this paper. First, JAC is a dynamic aspect system, which allows new aspects to be created, woven, and unwoven at runtime. Second, JAC can add and remove aspects on remote machines.

```
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[] arg = mi.getMethod().getArguments();  
10        Logger.add("Login: " + arg[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 4. A logging aspect written in JAC.

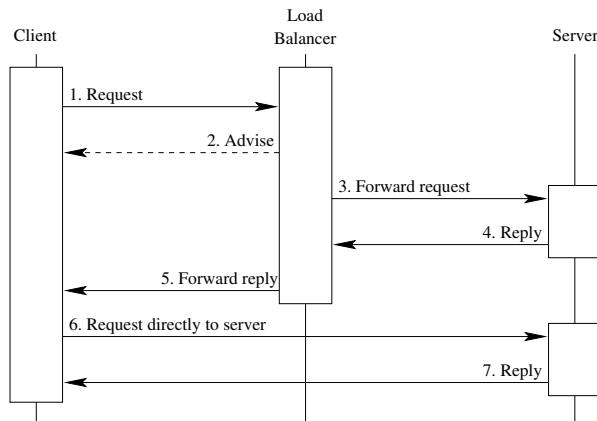
We will do this with the `remoteWeaveAspect()` and `remoteUnweaveAspect()` methods available in JAC. It is important to note that a process can unweave an aspect that was woven by another process.

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.

## 2.4 Advising Client Proxies in JAC

In our previous work, we used JAC to construct aspects at the server and weave them into the proxy at the client to create *smart proxies* [15]. The key to this earlier work was that aspects were passed from server to client, where most efforts in dynamic distributed aspects forward aspects from client to server. In addition, that work did not alter the underlying development model for creating Java RMI programs. That is, developers continued to use `rmic` to generate the proxy classes and the standard Java compiler to compile the application code. No new tools or extra development steps were added, and the client developer did not have to work with new libraries to develop smart proxies.

In that work, we applied aspects for caching and client-side input validation to client proxies to demonstrate the ideas. These two examples are optimizations to avoid remote method invocation if possible. For caching, return values for methods are stored in a hash table at the client and results are returned from the table on repeated calls. For client-side input validation, the arguments to a method were checked at the client to avoid sending a remote request for method calls with invalid arguments. However, neither example altered the communication structure of the application. Both eliminated remote calls that were unnecessary.



**Figure 5. Load balancing with dynamic remote aspects.**

This paper shows the more powerful capabilities of dynamic, distributed aspects, in particular their ability to alter the communication structure to distribute requests to a set of available servers. Importantly, the servers are in control of these aspects and can alter them as needed.

### 3 Load Balancing with JAC

Our approach to load balancing uses dynamic, distributed aspects forwarded from server to client to advise client-side proxy objects. These aspects allow us to use a balancer process to distribute clients to servers, but also allow the servers to shed load when necessary.

The overall process is shown in Figure 5. We start with what appears to be the standard solution using a balancer process, where clients initially send requests to the balancer. Indeed, the first request from each client is treated using this standard solution; the request is forwarded to a suitable server, and the results follow the reverse path. This is shown in messages 1, 3, 4, and 5. However, the balancer also dynamically weaves a JAC aspect on the client proxy while the request is being processed (message 2). This aspect alters the proxy to forward requests to a specific server. Thus, all subsequent requests are forwarded directly from client to the selected server (messages 6 and 7). Importantly, the balancer applies the aspect to the client when it is blocked awaiting the reply to its remote method invocation. This removes many of the potential concurrency problems that can arise in this type of system.

From here, the solution appears to execute like the second standard solution. A client is assigned to a given server and all requests are forwarded to the single location. Here, we exploit the ability to dynamically unweave an aspect in JAC. If the server is overloaded, it can unweave the aspect from the client proxy. After

this unweaving, the next request from that client is forwarded to the balancer process, which then applies a new aspect to forward client requests to another server. That is, once the aspect is unwoven, the scenario in Figure 5 repeats itself for the client, except that the client may be associated with a new server. Again, this unweaving is performed when the client is blocked awaiting a reply to reduce potential concurrency problems.

This style of load balancing addresses the limitations of the two schemes presented in Section 2.2. It avoids the need for the load balancer to be involved with each request, reducing its load and reducing the network latency of forwarding each request. It avoids the problem of clients caching information, since that information is now held in an aspect to which the client is oblivious. It also allows the server to shed load by unweaving aspects from clients, so they can be directed to new servers.

This style of load balancing is not without its own sources of overhead. The balancer must weave advice onto the client on the first request, though this can be mitigated by overlapping it with the execution of the request using a separate thread. Unweaving from the server has similar overhead. Advising a method adds overhead to its execution. JAC in particular adds overhead given its extensive use of Java Reflection, used to support weaving at runtime.

One desirable property of our strategy is that it lends itself to balancing client sessions, where a session is a series of individual requests that form one larger, logical request. It can be advantageous to forward all requests for a single session to the same server, but balance different sessions to different servers. In our strategy, the server can achieve this by simply unweaving advice from a client at the end of a session.

### 4 Overhead Measurements

In this section, we evaluate the overheads associated with the use of dynamic server-side aspects on load balancing. These overheads were measured using a set of microbenchmarks that consisted of three processes (client, balancer, and server) running on three separate machines connected by a 100 Mbit per second network. The client invokes a remote method that is handled using the strategy in Section 3. The remote method does not take any arguments, has no return value, and an empty method body, so the results presented here represent just the overhead introduced by the dynamic distributed aspects in JAC.

As a baseline, we measured the performance of the first load balancing strategy given in Figure 2, where all requests must be forwarded through the balancer before being directed to an appropriate server. From the per-

spective of the client, requests take an average of 0.51 ms per round trip. Note that this time represents a lower bound since the balancer manages a single server and does not have to make any decisions to balance the load. The request is simply forwarded to the server.

In assessing our load balancing strategy, we first assessed the overhead of the first request by the client, which requires the balancer to not only forward the request to the server but also weave advice into the client. From the perspective of the client, this first request takes an average of 5.64 ms. Note that we made sure to unweave the advice between successive calls to avoid any additional overhead that may accumulate by repeatedly weaving advice on the same join point. To reduce the overhead of this weaving, it is done by a separate thread that runs while the balancer forwards the method invocation to a server.

After the first request, the advice woven at the client proxy forwards all subsequent requests directly to the server without involving the balancer. This avoids the overhead of an extra exchange of messages, but incurs the overhead of invoking advice woven on the client proxy. Some aspect-oriented constructs can incur a significant performance penalty in ways that are not obvious [2]. In particular, around advice may include closure objects, which are expensive to create and use. The use of `proceed()` to invoke the code for a join point has unusual polymorphic properties that can be expensive to evaluate at runtime, and JAC uses Java Reflection which can be expensive. However, our balancing advice replaces the client proxy code rather than invoking it. Since `proceed()` is not called, some of this overhead should not be present.

To assess the overhead of the advice at the client proxy, we measured the cost of a remote method invocation with an advised proxy and the cost of normal remote method invocation directly from client to server. These two measurements give us the overhead of applying aspects to the client proxy. From the perspective of the client, the cost of a remote method with a balancing aspect was measured to average 0.357 ms per requests, which was identical to a round trip with an unadvised proxy. The advice adds no appreciable overhead to the execution of the proxy on the client.

Our aspect-oriented version, which allows requests to be sent directly to a server after the first request, cuts 0.153 ms from each request, a savings of 30%. However, this does come at the cost of weaving, which takes 5.64 ms in JAC. This requires clients to forward 44 requests to a given server to make up for this extra cost. Again, though, this represents a lower bound since our experimental setup does not include any scheduling decisions, computation, or significant data transfer for method ar-

guments and return values. In a more realistic environment, we expect this number to be lower. In particular, given that remote methods are normally large-grained to make up for communication overhead, this weaving process can be overlapped with the execution of the method and may impose little overhead in overall execution. For requests that benefit greatly from server affinity, this approach may be best even with its overhead.

When a server wishes to shed some of its load, it can unweave advice from a client proxy to force the proxy to forward its next request through the balancer. This requires an extra message from server to client while the remote method is executing. We can overlap this unweaving process with the execution of the remote method, just as we did when weaving the advice in the balancer. From the perspective of the client, a request that involves this unweaving takes an average of 8.14 ms. It must be noted that our tests have an empty method body, so there is no overlap between method execution and unweaving. As a result, this number is the worst case. Again, unweaving can be overlapped with method execution to hide some of the cost.

A final concern is whether unweaving advice from a client leaves any residual code behind that adds overhead. Earlier benchmarks from [15] found no such residual overhead. When advice is unwoven, the client code performs as it did before the advice was woven.

## 5 Related Work

Related work with respect to load balancing in Java RMI was covered in Section 2.2. This section focuses on other research in distributed AOP.

DJcutter allows developers to create pointcuts that specify both a specific event in the execution of a program and the host on which that event must take place [11]. These pointcuts are automatically distributed to object servers. However, all advice is run on a separate *aspect server* rather than on the host that triggered the pointcut. Similarly, D supports distribution in its aspects but also cannot distribute the execution of advice [7]. Because of this, neither is suitable for this work.

FORMI is built on a *fragmented object model*, where an object implementation is split across fragments that can be individually distributed across a network [4]. 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. FORMI provides the same flexibility as our aspect-oriented approach, and also does not impact client code. However, FORMI introduces its own stub compiler, changing the development process.

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 [14]. 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.

AWED provides a more flexible model that supports the distribution of pointcuts and advice separately [10]. A pointcut can specify the host on which a join point applies and the host on which advice should be run. AWED permits advice to run on several hosts, which makes it useful for replication consistency mechanisms. This research could have used AWED rather than JAC. One potential problem with the use of AWED is that it uses new language constructs to specify pointcuts, and it does not appear that these constructs can be parameterized with runtime arguments. The `pointcut()` method in JAC does permit such parameters, making it more flexible.

Security concerns are a serious issue in this dynamic aspect-oriented approach since we are allowing one process to alter the code of another. 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 to allow the server to advise it. Finer-grained control is not possible, though there are several proposals on how to address this problem [6, 12].

## 6 Conclusions

In this paper, we presented a dynamic, aspect-oriented implementation of load balancing in Java RMI. Initial requests from a client are directed to a balancer process, which forwards the request to a server while simultaneously weaving an aspect on the client proxy. The woven aspect instructs the client to forward all subsequent requests to a specific server. However, if that server needs to shed some of its load, it can unweave the aspect to force the client to find another server. This approach reduces the overhead of having all requests forwarded by a balancer process but provides a more dynamic ability to redistribute load when necessary. In addition, all decisions are made by the server based on application needs.

To evaluate this approach, we presented results from a series of microbenchmarks. These results show that the client does not suffer appreciable overhead from the advice woven into its proxy, and that the cost per request does fall by 30% by sending requests directly to a server without an intervening balancer process. However, the

cost of weaving and unweaving advice is high and must be amortized over a number of calls from client to server.

In the future, we could improve this research by augmenting the underlying Java RMI protocol to piggyback aspects and advice with reply messages. Our current implementation tries to overlap the transmission and weaving of advice with the execution of remote methods using threads. This augmented protocol would require clients to be more aware of the new functionality, but would reduce the overhead in applying advice over the network and obviate the need for extra threads.

## References

- [1] Apache HTTP Server Project. *URL Rewriting Guide*, 2008. <http://httpd.apache.org/docs/2.2/misc/rewriteguide.html>.
- [2] B. Dufour *et al.* Measuring the dynamic behaviour of AspectJ programs. In *Proceedings of the 19th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 150–169, 2004.
- [3] jGuru. *Remote Method Invocation: Tutorial and Code Camp*, 2000. <http://java.sun.com/developer/onlineTraining/rmi/RMI.html>.
- [4] R. Kapitza, J. Domaschka, F. Hauck, H. Reiser, and H. Schmidt. FORMI: Integrating adaptive fragmented objects into Java RMI. *IEEE Distributed Systems Online*, 7(10), 2006.
- [5] G. Kiczales *et al.* Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, volume 1241 of LNCS, pages 220–242. Springer-Verlag, 1997.
- [6] D. Larochelle *et al.* Join point encapsulation. In *Proceedings of the 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] H. Meling *et al.* Jgroup/ARM: A distributed object group platform with autonomous replication management. *Software: Practice and Experience*, 2008. To appear.
- [9] N. Narasimhan *et al.* Interceptors for Java remote method invocation. *Concurrency and Computation: Practice and Experience*, 13(8-9):755–774, 2001.
- [10] L. Navarro *et al.* Explicitly distributed AOP using AWED. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 51–62, 2006.
- [11] M. Nishizawa *et al.* Remote pointcut: A language construct for distributed AOP. In *Proc. 3rd International Conference on Aspect-Oriented Software Development*, pages 7–15, 2004.
- [12] H. Ossher. Confirmed join points. In *Proceedings of the 2006 Workshop on Software Engineering Properties of Languages for Aspect Technologies*, 2006.
- [13] R. Pawlak *et al.* JAC: An aspect-based distributed dynamic framework. *Software: Practice and Experience*, 34(12):1119–1148, 2004.
- [14] N. Santos *et al.* A framework for smart proxies and interceptors in RMI. In *Proceedings of the 15th ISCA International Conference on Parallel and Distributed Computing Systems*, 2002.
- [15] A. Stevenson and S. MacDonald. Smart proxies in java rmi with dynamic aspect-oriented programming. In *Proceedings of the 2008 International Workshop on Java and Components for Parallelism, Distribution and Concurrency*, 2008.
- [16] Sun Microsystems, Inc. *Dynamic Proxy Classes*, 2004. <http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html>.