Interceptors for Java Remote Method Invocation*

Nitya Narasimhan, L. E. Moser and P. M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106

Abstract

Interceptors are software mechanisms that are inserted into an application at runtime, and can subsequently provide hooks to introduce new services to the application in a transparent manner. We have developed three distinct interceptors that are targeted at Java Remote Method Invocation. Our interceptors are novel in that they are developed independent of the application, and are easy to deploy, requiring little or no modification to the application. We exploit interception in the Aroma System that we have developed, to install mechanisms for the consistent replication of Java objects at runtime. These replication mechanisms can be exploited to provide fault tolerance and high availability to a distributed application, in a transparent manner.

1 Introduction

The Distributed Java Object Model [12] provides support for object serialization, distributed garbage collection, dynamic classloading and remote method invocation mechanisms to ease the development of distributed Java applications. However, the model lacks the support required to ensure that these applications operate in a reliable and highly-available manner. As a result, if continuous, fault-tolerant operation is desired, the developer must develop and integrate the necessary fault tolerance mechanisms into the application.

This approach has several disadvantages. Most application developers do not possess the skills or the expert knowledge to implement complex fault tolerance mechanisms correctly. Furthermore, developing these mechanisms, and integrating them into every application that requires them, is a difficult and inefficient process. The ideal solution is to have fault tolerance experts develop the necessary mechanisms and to make them available to the distributed application in a transparent manner.

An *interceptor* is a software device that can be introduced at runtime into the communication path of a distributed application, and can subsequently be exploited to analyze, modify, or reroute, the intercepted communication. Such interceptors can be used to facilitate replication by routing the intercepted messages to replication middleware. The middleware can subsequently route the message to one or more replicas of the intended destination, and can implement the infrastructure and mechanisms required to ensure replica consistency. Thus, by taking advantage of interception, replication mechanisms can be deployed at runtime, and can be exploited by distributed applications for reliable, highly-available operation.

In this paper, we describe the design and implementation of three distinct interception mechanisms for Java Remote Method Invocation (Java RMI)[11], and the advantages and disadvantages of each

*This research has been supported by AFOSR MURI under Contract F49620-00-1-0330.

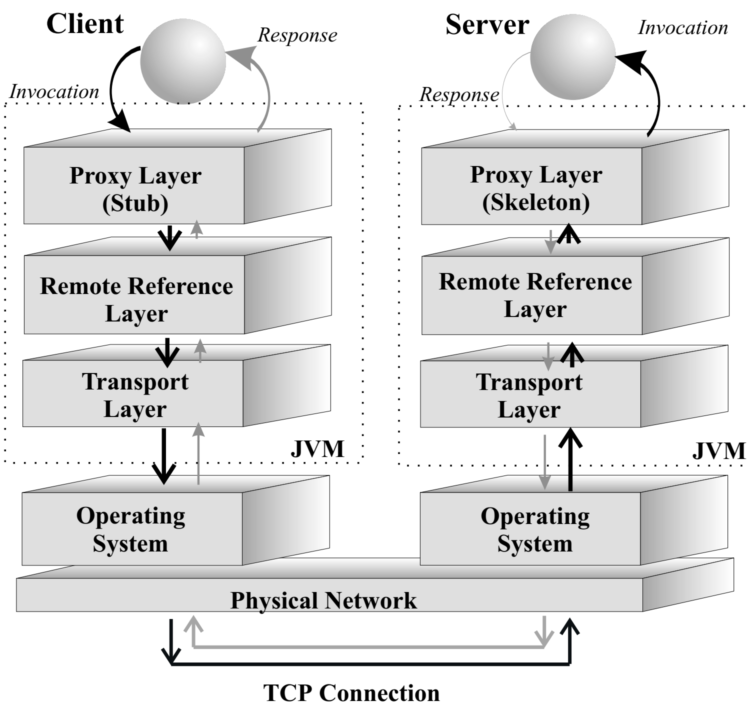

Figure 1: The JRMI Architecture.

of these mechanisms. Preliminary performance measurements are provided to evaluate the overhead associated with each approach. We also discuss the design of the Aroma System, where the interception mechanism is used to enhance Java RMI with mechanisms for the consistent replication of Java RMI objects.

2 The Java RMI Architecture

The interception mechanisms that we have developed are targeted at the Java RMI model, with the deployment of the interceptor being dependent on functionality that is specific to this model. Figure 1 shows an overview of the Java RMI model, illustrating its three-tier protocol stack.

A Java RMI server object must implement at least one `java.rmi.Remote` interface to identify its ability to service remote method invocations. The set of `Remote` interfaces implemented by the server defines the list of methods that can be invoked on it by a remote client. The server *reference* – consisting of a TCP/IP endpoint, and an object identifier unique to that host – helps to identify the server uniquely within the distributed system. Every Java RMI server is also associated with a stub that implements the same set of `Remote` interfaces, and contains a copy of the server reference.

A client wishing to invoke methods on this server must retrieve a copy of the server stub, and install the stub on its local JVM. A *rmiregistry* process facilitates this bootstrapping by maintaining a transient database of stringified server names that map to the corresponding server stubs. An active server can `bind` its stub to a well-known name in the registry. Subsequently, a client equipped with this name can `lookup` the registry to retrieve the stub. Once bootstrapping is completed, the registry plays no further role in the client-server communication.

A JRMI operation is initiated by a client making an invocation on the installed server stub; the stub uses the server reference to establish a TCP/IP connection to the server, and forwards the request parameters. The request is delivered to the server-side skeleton, which acts as a client proxy, invoking the request locally on the server, retrieving the response, and returning this response to the stub. Subsequently, the stub uploads the result to the client. Thus, the interaction between stub and skeleton masks all remote communication from the application.

The Java2 Standard Edition (J2SE) supports two implementations of the JRMI architecture, namely, RMI-JRMP and RMI-IIOP. The RMI-JRMP implementation uses the Java Remote Method Protocol (JRMP) [11], an indigenous TCP/IP-based protocol with simple semantics. JRMP exploits Java-specific mechanisms and, thus, can be used only for pure Java client-server applications. The RMI-IIOP model adopts the Internet Inter-ORB Protocol (IIOP), native to the CORBA [9] standard, as its underlying transport protocol. By exploiting IIOP, RMI-IIOP facilitates communication between *modified* RMI-JRMP objects and CORBA objects implemented in languages such as C++. In this paper, all interception mechanisms are designed for the RMI-JRMP model, unless specified otherwise.

3 Interception

An interceptor is a software device that can be deployed at runtime, and subsequently exploited to introduce new functionality to an executing application. In the Aroma System described in Section 7.2, we exploit interceptors to enhance the Java RMI model with the support required for transparent replication of Java RMI objects. For this purpose, the basic function of the interceptor is to capture the parameters of a Java RMI invocation. By embedding custom code within the interceptor, we can subsequently analyze, modify or enhance the request; alternatively, we can reroute the invocation to one or more replicas of the intended destination. Based on the functionality of this custom code, we can operate an interceptor in one of three modes:

- *Read-only* mode for analyzing intercepted messages
- *Enhancement* mode for augmenting the content of the intercepted messages
- *Transformation* mode for modifying the existing content of the intercepted messages.

By selecting the mode of operation, we can exploit the interceptor for different purposes. For instance, read-only interceptors can log invocations, enhancement interceptors can append contextual information to the message, and transformation interceptors can execute encryption algorithms to enforce privacy.

Read-only interceptors do not modify default application behavior. However, enhancement and transformation interceptors have the potential to introduce behavior that is visible to the application, usually an undesirable consequence. Thus, we enforce a simple rule in using interceptors; for every enhancement or transformation interceptor in the system, there must exist a complementary interceptor that reverses the enhancement or modification, respectively.

3.1 Interception Points in the Java RMI model

An *interception point* identifies locations along the Java RMI communication path at which an interceptor can be successfully deployed; the interception point also decides the content and significance of the intercepted data. Typically, the interception point can occur within one of the three layers of the Java RMI protocol stack, namely the proxy layer, the remote reference layer and the transport layer. Because all Java RMI objects communicate over TCP/IP, additional interception points can also be identified at the Java networking infrastructure, allowing Java RMI messages to be intercepted at a much lower level in the communication path.

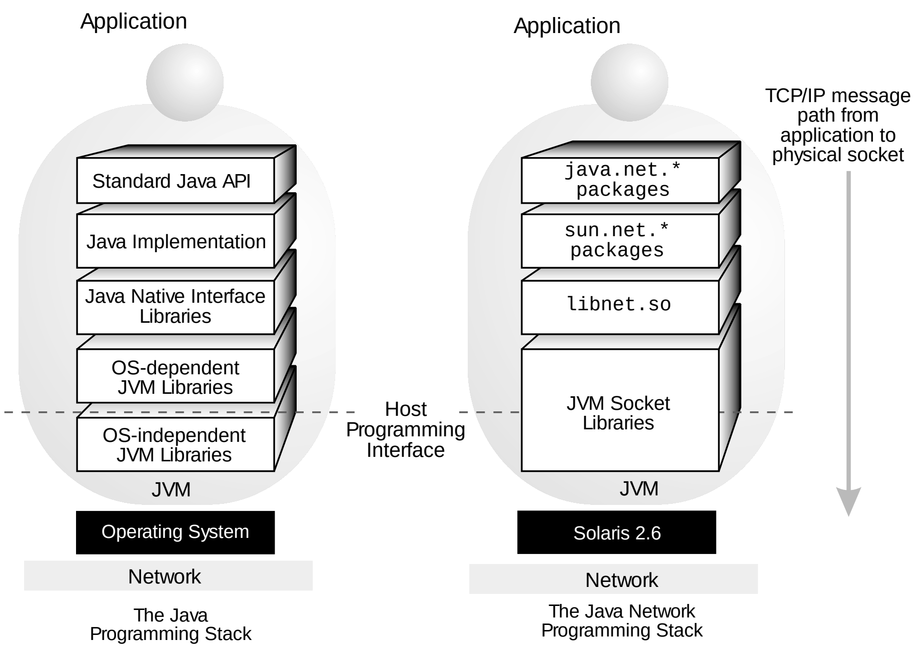
Application
Standard Java API
Java Implementation
Java Native Interface Libraries
OS-dependent JVM Libraries
OS-independent JVM Libraries
JVM
Operating System
Network
The Java Programming Stack
Host Programming Interface
Application
java.net.* packages
sun.net.* packages
libnet.so
JVM Socket Libraries
JVM
Solaris 2.6
Network
The Java Network Programming Stack
TCP/IP message path from application to physical socket

Figure 2: The `java.net` support infrastructure.

Figure 2 shows the implementation hierarchy for a standard Java API, and identifies the specific implementation levels that correspond to the standard Java networking API. An interceptor at the networking layer could be located at one of the following levels:

- The `java.net` level. This level contains networking classes implemented in Java, that use standard APIs. Interceptors at this level are born portable.
- The `sun.net` level. This level contains networking classes implemented in Java, and uses hidden, non-standard APIs. Interception points that exploit features at this level cannot be guaranteed the same support in subsequent releases of the JDK.
- The `libnet` level. This level contains the Java Native Interface (JNI) code that provides the glue between the Java classes and the underlying native libraries. Interceptors at this level will have standard interfaces to exploit, but will need to deal with portability issues.
- The `JVM Socket Libraries` level. This level contains native libraries implemented at the JVM level. These libraries contain code that is either independent of the operating system, or that maps onto operating system-specific functionality.

The existence of an interception point at any level is decided by three factors. First, the availability of some "hook" at that level, that can be exploited to introduce the interception code at runtime. Second, the performance penalty incurred by introducing interception code at that level, and finally, the portability of the interceptor mechanism. Based on the first of these factors, we have identified three possible approaches to developing an interceptor for the Java RMI model. The second and third factors allow us to compare the different approaches for building interceptors, and, consequently, to select the approach that is most appropriate to the application at hand.

The three approaches to building interceptors are based on specific interception "hooks", namely, the Dynamic Proxy, the RMISocketFactory and library mediation. The Dynamic Proxy approach

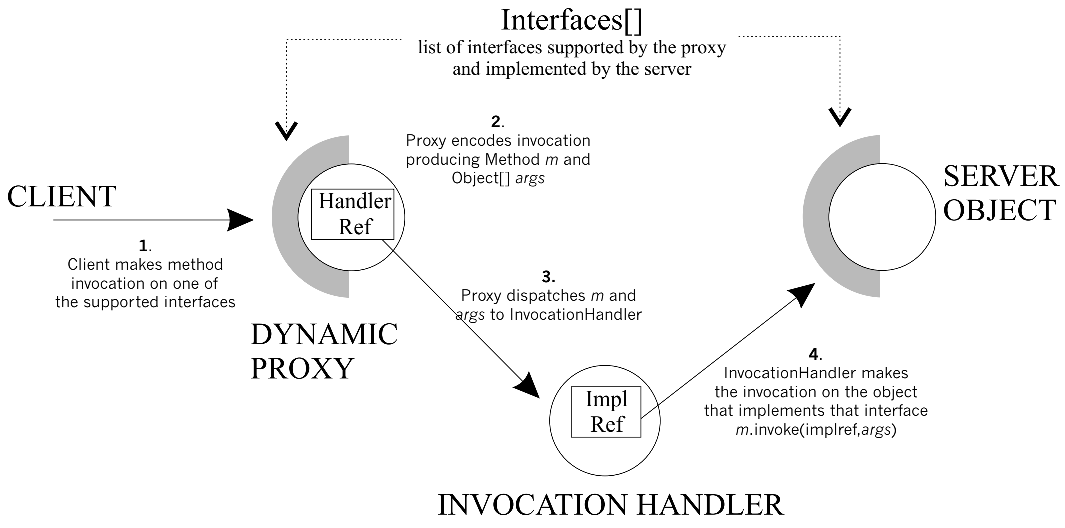
Interfaces[]
list of interfaces supported by the proxy and implemented by the server
CLIENT
1. Client makes method invocation on one of the supported interfaces
2. Proxy encodes invocation producing Method m and Object[] args
Handler Ref
DYNAMIC PROXY
3. Proxy dispatches m and args to InvocationHandler
SERVER OBJECT
Impl Ref
4. InvocationHandler makes the invocation on the object that implements that interface m.invoke(implref,args)
INVOCATION HANDLER

Figure 3: The Dynamic Proxy mechanism.

places the interception point at the proxy layer of the Java RMI protocol stack, the RMISocketFactory approach achieves interception at the `java.net` level of the networking stack, and the Library Mediation approach performs interception at the JNI level of the networking library, as represented by the `libnet` library.

4 The Dynamic Proxy Approach

The concept of dynamic proxies was introduced in release 1.3 of the Java2 Standard Edition, and exploits the reflection mechanism supported by Java. A Dynamic Proxy class is by definition "a class that implements a list of interfaces specified at runtime such that a method invocation through one of the interfaces on an instance of the class will be encoded and dispatched to another object through a uniform interface". This mechanism can be used to create, at runtime rather that at compile-time, a proxy for any object that implements the specified list of interfaces.

The Dynamic Proxy class is associated with an `InvocationHandler` object that implements a single `invoke` method, as shown in Figure 3. When a method invocation occurs on an instance of the Dynamic Proxy class, the parameters of the invocation – namely the method and the arguments – are encoded, using reflection, into a `java.lang.reflect.Method` object (that identifies the method), and an array of objects containing the values of the arguments. The Dynamic Proxy instance dispatches these encoded parameters of the invocation to the `InvocationHandler`; the value returned by the `InvocationHandler` is subsequently returned as the result of the invocation.

The `InvocationHandler` is an interface defining a single `invoke` method that takes, as parameters, a proxy instance, a `Method` object and an array of arguments. This interface must be implemented by any "delegate" class that wishes to register itself as the invocation handler for a Dynamic Proxy instance. This delegate class can either provide a concrete implementation of the list of interfaces defined for the Dynamic Proxy, or can, in turn, delegate the invocation to one or more objects that provide concrete implementations of that method.

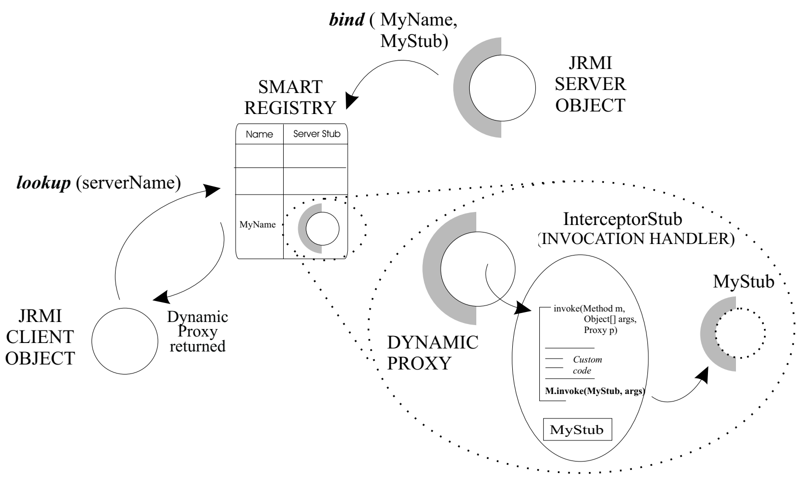

Figure 4: The Dynamic Proxy approach to interception.

4.1 Design

This functionality of the `InvocationHandler` allows it to separate the invocation interface from the implementation of the method that executes the invocation; this is the “hook” that we exploit for interception. Any custom code inserted at the `InvocationHandler` level can be used to analyze or modify the invocation parameters, or to reroute the invocation to one of many distinct objects that provide a concrete implementation of the invoked method. Therefore, by generating a Dynamic Proxy for a Java RMI server, and creating a custom `InvocationHandler`, we can intercept all invocations destined for that server.

In this case, the set of interfaces implemented by the Dynamic Proxy must correspond to the set of `Remote` interfaces implemented by the Java RMI server. Further, the `InvocationHandler` implementation must contain a reference to the server object, to facilitate the invocation of methods on that object. Because we are dealing with remote objects, the server reference in this case corresponds to the Java RMI server reference, and consists of a TCP/IP endpoint and an object identifier. With the existing Java RMI APIs, there are no methods to obtain this server reference directly; however, we can obtain a copy of the server stub (which holds a copy of this reference), and make invocations on the stub rather than directly on the server.

4.2 Implementation

Our simple interceptor mechanism consists of a Dynamic Proxy generated for the Java RMI server stub as illustrated in Figure 4. If, in a future release of the Java2 Standard Edition, standard API methods can be exploited to retrieve a copy of the server reference directly, then the interceptor mechanism will consist of a dynamic proxy generated directly for the Java RMI server object

The InterceptorStub implements the InvocationHandler interface, and maintains a handle to the stub for the server object. The dynamic proxy is created for the list of `java.rmi.Remote` interfaces supported by the stub, and, as a result, by the server. An invocation made on the Dynamic Proxy is encoded into the corresponding `Method` and arguments format, and dispatched to the InterceptorStub. Custom code can be inserted at this point to process the invocation parameters before actually invoking the method on the server stub. In the simple example shown, the custom code logs the invocation parameters before delegating the invocation.

To deploy this interceptor, we need to ensure that the client uses the Dynamic Proxy in place of the standard server stub. Further, to maintain transparency to the application, we need to achieve this without modifying application code. Our solution is to use a custom SmartRegistry instead of the standard *rmiregistry*, as illustrated in Figure 4. The SmartRegistry maintains a transient database that maps server names onto a Dynamic Proxy for that server, rather than onto the server stub. When the server invokes the `bind` method on the SmartRegistry, it passes as parameters, its standard stub and a stringified server name. The SmartRegistry subsequently creates an InterceptorStub instance using the standard stub, and creates a Dynamic Proxy instance with this InterceptorStub as the invocation handler. This Dynamic Proxy is bound against the stringified server name in the SmartRegistry, in place of the standard server stub. A client that looks up the registry by server name will retrieve, and install, a copy of this Dynamic Proxy by default. Subsequently, all invocations made by the client are dispatched through the proxy, and can thus be intercepted by our mechanisms.

5 The RMISocketFactory Approach

An alternative approach to interception exploits the RMISocketFactory mechanism, and places the the interception point within the transport layer of the Java RMI protocol stack. The abstract `java.rmi.server.RMISocketFactory` class defines methods to create TCP/IP client and server sockets for the RMI-JRMP runtime. A standard implementation of this interface, which we refer to as the *master socket factory*, is installed as the default socket factory at runtime. The master socket factory creates instances of `java.net.Socket` and `java.net.ServerSocket`, in response to `createSocket()` and `createServerSocket()` requests, respectively, from the RMI-JRMP runtime. The default set-up is illustrated in Figure 5(a).

The `java.net.Socket` object implements the Java interface to the physical socket; it has an associated `InputStream` object and an `OutputStream` object that facilitate read and write operations, respectively, on the socket, as shown in Figure 5(b). An application can provide an alternative implementation of the `RMISocketFactory` interface and install this version as the JVM-wide socket factory for the RMI-JRMP model, supplanting the default master socket factory. Our Interceptor comprises the following classes.

- An *InterceptorSocketFactory* that implements
  `java.rmi.server.RMISocketFactory`
- An *InterceptorSocket* that extends
  `java.net.Socket`
- An *InterceptorServerSocket* that extends
  `java.net.ServerSocket`
- An *InterceptorOutputStream* that extends
  `java.io.InputStream`
- An *InterceptorInputStream* that extends
  `java.io.OutputStream`

At runtime, the InterceptorSocketFactory instance caches a handle to the existing default master socket factory, and installs itself as the new default socket factory. The InterceptorSocketFactory returns instances of InterceptorSocket and InterceptorServerSocket, in response to `createSocket()` and `createServerSocket()` requests, respectively. The latter classes constitute adapters [3] that internally

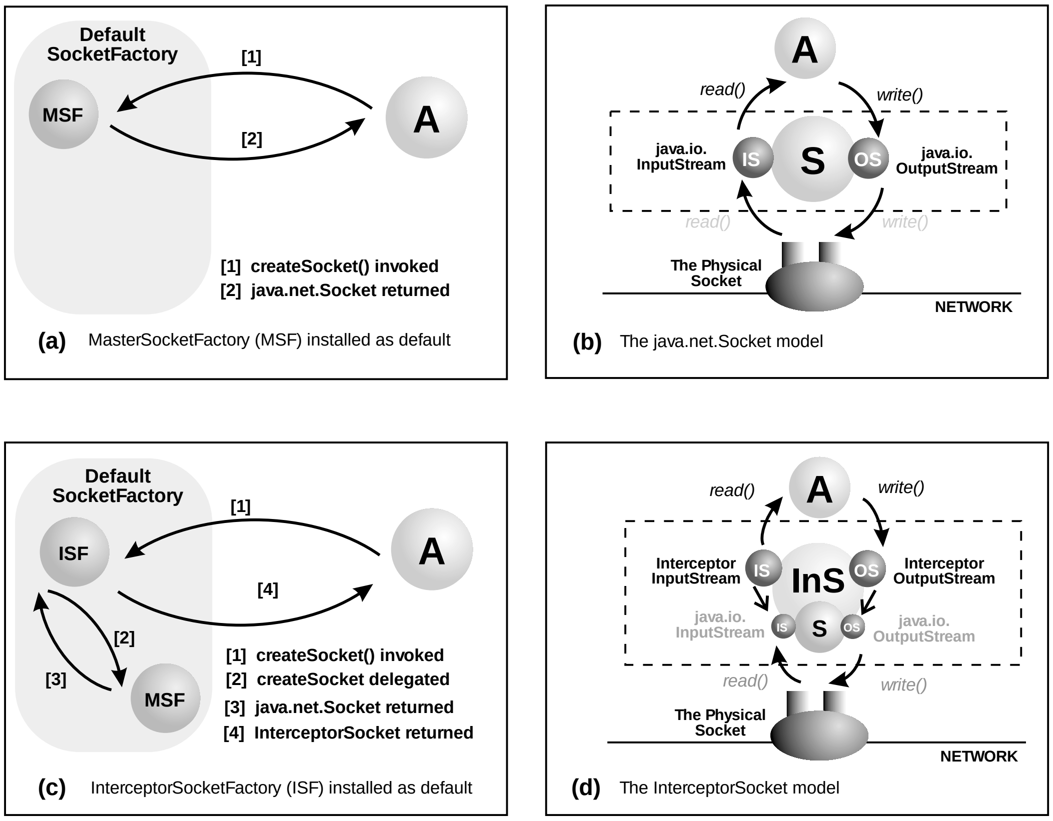
Default SocketFactory
MSF
A
[1]
[2]
[1] createSocket() invoked
[2] java.net.Socket returned
(a) MasterSocketFactory (MSF) installed as default
A
read()
write()
java.io. InputStream
IS
S
OS
java.io. OutputStream
read()
write()
The Physical Socket
NETWORK
(b) The java.net.Socket model
Default SocketFactory
ISF
A
[1]
[4]
[2]
[3]
MSF
[1] createSocket() invoked
[2] createSocket delegated
[3] java.net.Socket returned
[4] InterceptorSocket returned
(c) InterceptorSocketFactory (ISF) installed as default
read()
A
write()
Interceptor InputStream
IS
InS
OS
Interceptor OutputStream
java.io. InputStream
IS
S
OS
java.io. OutputStream
read()
write()
The Physical Socket
NETWORK
(d) The InterceptorSocket model

Figure 5: The RMISocketFactory approach.

invoke the appropriate create() method on the cached socket factory to obtain a standard `Socket` object to which they delegate calls at runtime. Figure 5(c) illustrates the interceptor configuration and behavior. The Interceptor socket classes contain InterceptorInputStream and InterceptorOutputStream objects to facilitate read and write operations, respectively, on Interceptor sockets. These interceptor stream objects are built over the corresponding streams associated with the underlying `Socket` instance, as shown in Figure 5(d).

The InterceptorInputStream and InterceptorOutputStream objects are conduits between the application and the streams associated with the underlying physical socket. Because all TCP/IP messages generated by the application must pass through these objects, we can place custom code within them to implement our Interceptor. The socket factory is inherently portable, being implemented completely in Java. However, this mechanism is more suitable for RMI-JRMP applications than for RMI-IIOP objects. Further, the introduction of the interceptor is not achieved in a transparent manner. Interceptor activation requires the addition of a single `setSocketFactory()` call within the application's `main()` method, at the point where the Java Virtual Machine is being initialized. However, the interception mechanisms themselves are invisible to the application.

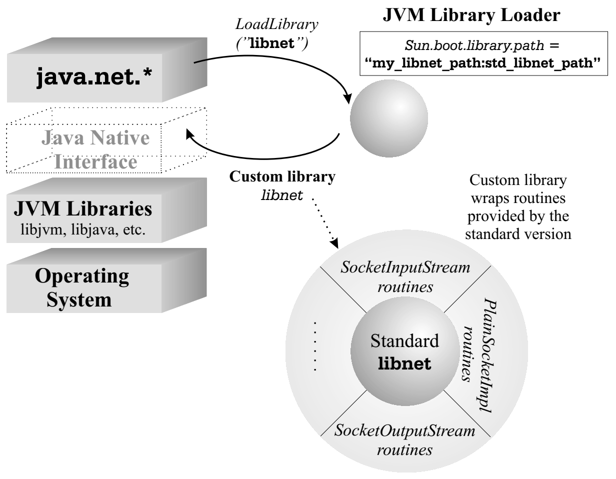

Figure 6: The Library Mediation Approach.

6 The Library Mediation Approach

The socket factory approach described in Section 5 introduced the interceptor at the Java API level using standard Java mechanisms. Our second approach introduces the interceptor at a lower level in the implementation hierarchy; in this case, however, the interceptor is activated in a manner that is completely transparent to the application.

We exploit the fact that the JVM uses the services of the *Java library loader* to load the native libraries requested by a Java application, at runtime. The loader searches for libraries in locations specified by the `sun.boot.library.path` property; the first library encountered that matches the request is loaded into the Java runtime. Therefore, by implementing a custom version of the native library, and pre-pending its location to `sun.boot.library.path`, we "trick" the loader into finding our custom version ahead of the standard native library. We refer to this approach as *library mediation.*

The Java networking model employs two sets of native libraries, one at the JNI level, and the other at the JVM level. Mediation on one of these libraries gives us the hook that we need to introduce our interceptor code transparently; this approach entails no modification to the application code. We elected to mediate the JNI `libnet` library, rather than the JVM socket libraries, for two reasons:

- The JVM socket libraries contain operating system specific code. Implementing a portable interceptor would require maintaining versions of the interceptor for every operating system that supports the JVM. The JNI library interfaces are more consistent across JVMs; the method signatures in `libnet.so` are easily derived from the native method declarations provided in the standard `java.net` package. Because the `java.net` code is unchanged across JVMs, the `libnet` method signatures remain the same; porting the interceptor requires recompilation of the same code for different operating systems.

- The term "JVM socket libraries" represents a logical collection of routines implemented within larger native libraries at this level. Substitution of a larger library merely to intercept calls to a subset of its routines is impractical, and impossible to achieve without access to the source code. The `libnet` library, on the other hand, contains all of the routines used by `java.net` classes, and implements no other functionality. Because the `libnet` library is essentially a JNI wrapper, implementing a custom version is considerably easier.

The first step in building the interceptor is to deduce the definition of the `libnet` library interface. The JDK contains a `javah` utility that operates on a Java class and generates a native header file; this file contains the JNI signatures of all native methods declared in that class. Because JNI routines use fully-qualified names, an analysis of the symbol table of `libnet` provides an accurate assessment of the classes that declared the JNI routines; running the `javah` utility on those classes enables us to discover the JNI signatures that we need to define within our `libnet` library. For convenience, we will use `libnet` to refer to the standard version of this library, and *libnet* to refer to our custom implementation.

Our *libnet* implementation provides the ideal location to embed our interceptor code. We develop *libnet* as a collection of wrappers around the routines of the standard `libnet` implementation; by default, all calls on the *libnet* wrapper are delegated to the corresponding implementation of those calls in the `libnet` version. Because we are interested in intercepting only TCP/IP communication, we introduce code within the TCP/IP-related routines in *libnet* to perform pre-delegation and post-execution processing for those routines. This code can enhance or modify the messages handled by the `libnet` routines, and provides the foundation for our interceptor implementation. This interceptor approach does not provide the inherent portability of the Java-based interceptor described in Section 5 due to its use of native JNI code. However, the portability of our interceptor is not as issue because the interceptor lies within the framework of the JVM; the portability offered by the JVM is implicitly extended to the interceptor.

7 Exploiting Interception to Provide Transparent Fault-Tolerance

Replication is a well-known solution for providing fault-tolerant, highly-available operation. However, this raises a number of issues, depending on the type of object being replicated, and the replication scheme enforced.

7.1 Simple Failover using the Dynamic Proxy Approach

In a simple case, stateless Java RMI server objects can be replicated for availability. Because the state of the server object is not modified as the result of an invocation, maintaining a consistent state across replicas is relatively simple. Further, because the objects are replicated for availability, we can choose to route distinct client requests to different server replicas based on the current load at each replica. In this context, a measure of fault-tolerance can be provided by using a simple fail-over mechanism. The invocation is forwarded to any one of the available replicas; in the event that the replica fails, the mechanisms detect the fault, select a live server replica and retry the invocation, thus concealing the fault from the client. Because the replicas are stateless, we do not require complicated mechanisms to ensure that the state of all replicas is maintained consistent.

The Dynamic Proxy interceptor can be exploited to support this replication scheme. Each replica of the stateless server binds itself to the same name in the SmartRegistry. The InterceptorStub (which implements the InvocationHandler interface) is created for the first such bind request, and a Dynamic Proxy registered for that stringified name. The InterceptorStub is equipped to hold an array of server stubs, each corresponding to a replica of the same server object. When a second replica invokes the bind method with the same stringified name, the SmartRegistry determines that a Dynamic Proxy is currently registered for that name; the stub is subsequently added to the array maintained by the InvocationHandler for that proxy. A client that looksup the SmartRegistry will retrieve a Dynamic Proxy and its associated InvocationHandler.

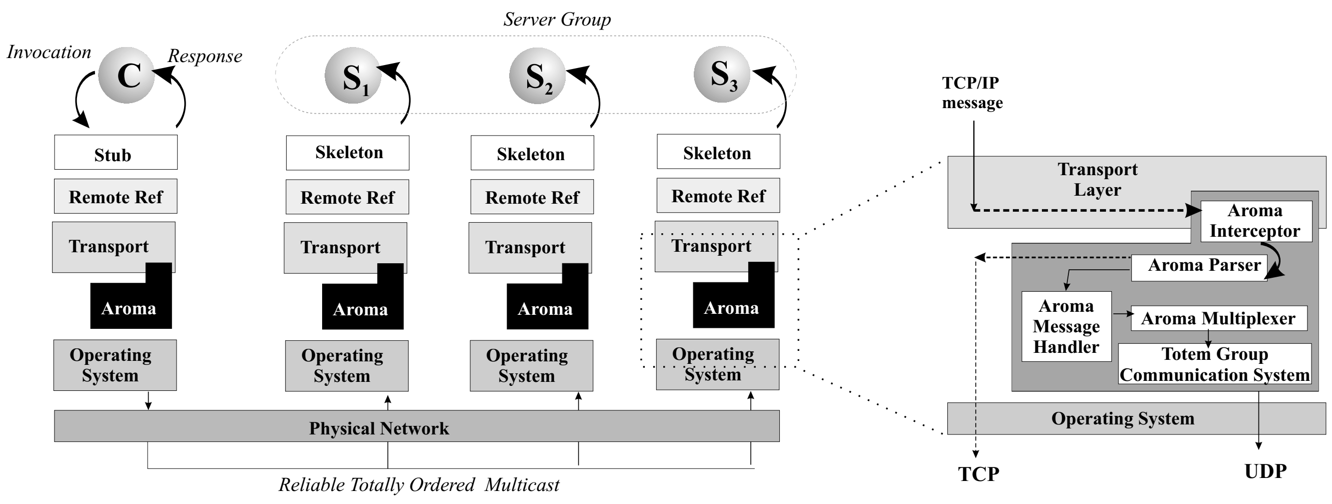

Figure 7: The Aroma System Architecture.

Simple fail-over mechanisms can be implemented as custom code within the InvocationHandler. An invocation dispatched to the handler is invoked on the first stub in the array maintained by the handler. A server fault, detected by a `java.io.IOException`, can be caught at the InvocationHandler using a simple try-catch block around the implementation of the *invoke* method. At this point, the handler can retrieve a second stub from its local array, and retry the invocation, thereby masking the server fault from the client.

7.2 Consistent Replication Using the Aroma System

The more difficult case involves replicating stateful objects, and supporting additional replication schemes such as active and passive replication. In this context, every invocation is typically forwarded to all replicas of the intended server, and every generated response is returned to all replicas of the client that made the invocation. More complex mechanisms are required to guarantee that all replicas of a given Java RMI object are maintained consistent at all times. Some of these issues are discussed in [7].

The Aroma System that we have developed is middleware that exploits interception to enhance the Java RMI model with support for replication, in a transparent manner. Both client and server objects are replicated, with little or no modification of the application code. This section provides a high-level overview of the design of the Aroma System; more details can be found in [6, 7]

The Aroma System adopts the object group paradigm [4] for transparent replication of JRMI objects. All replicas of an object form an object group, and are represented by the `Remote` interface associated with the object. To achieve replica consistency, all replicas (members) of a replicated object (object group) must "see" the same sequence of messages in the same order; thus, they will perform the same operations, resulting in the same state being maintained across all of the replicas. The Aroma system exploits the services of a reliable totally-ordered multicast group communication system, such as Totem [5], for communication between and within object groups. The reliable total ordering of messages is crucial to achieving replica consistency in an efficient manner.

Figure 7 gives an overview of the Aroma System, and highlights three main components: the Aroma Interceptor, the Aroma Parser and the Aroma Message Handler.

- The *Aroma Interceptor* is based on the library mediation approach, and resides at the transport

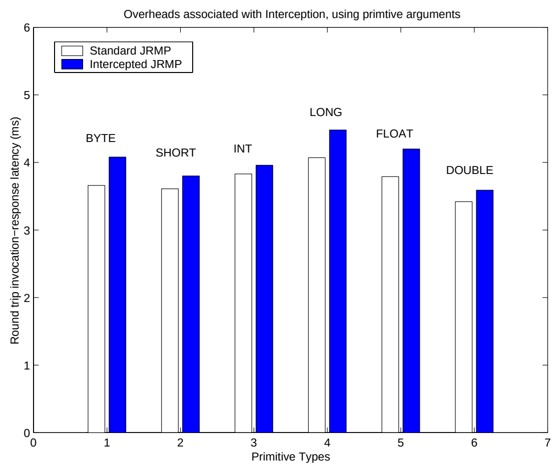

Figure 8. Latency introduced into RMI-JRMP applications by the Dynamic Proxy Interceptor, for primitive arguments.

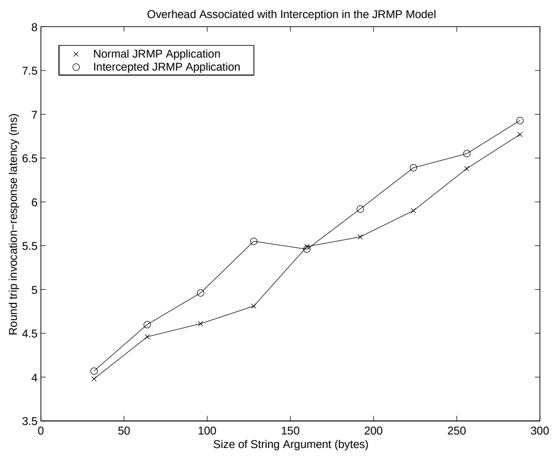

Figure 9. Latency introduced into RMI-JRMP applications by the Dynamic Proxy Interceptor, for string arguments.

layer of the Java distributed object model. It captures networking calls made by the application, including *read* and *write* calls required to receive and send TCP/IP messages, respectively. By handling the *reads*, the Interceptor can manipulate inbound messages to the application; similarly, by handling the *writes*, it can control the format and content of outbound messages generated by the application. Because Aroma intercepts *every* TCP/IP call made by the application, the Aroma Parser is needed to filter those TCP/IP messages that conform to known JRMI formats. All valid JRMI messages are forwarded to the Aroma Message Handler; all other TCP/IP messages are released, and continue their progress along the default TCP/IP path.

- The *Aroma Message Handler* achieves two important functions; it adapts the intercepted TCP/IP messages for multicast over the group communication system, and it performs the mapping between group-specific identifiers and the corresponding local replica's identifiers. In this context, the Message Handler constitutes the boundary that separates group-level communication from object-level communication.

- The *Aroma Multiplexer* provides the interface to the underlying multicast protocol. It encapsulates the adapted JRMI message, along with an Aroma-specific header, into a message suitable for multicast over the reliable totally-ordered multicast protocol.

By exploiting the Interceptor, the Aroma System introduces these mechanisms *transparently* to JRMI, thereby enhancing it with the basic support required for replication. By exploiting an efficient multicast protocol for communication between replicated objects, the Aroma Interceptor provides higher performance for fault-tolerant Java applications than could be obtained using multiple TCP/IP connections.

8 Interceptor Performance

To determine the feasibility of exploiting interception for our replication mechanisms, we evaluated the overheads associated with introducing the Interceptor into standard RMI-JRMP applications, and

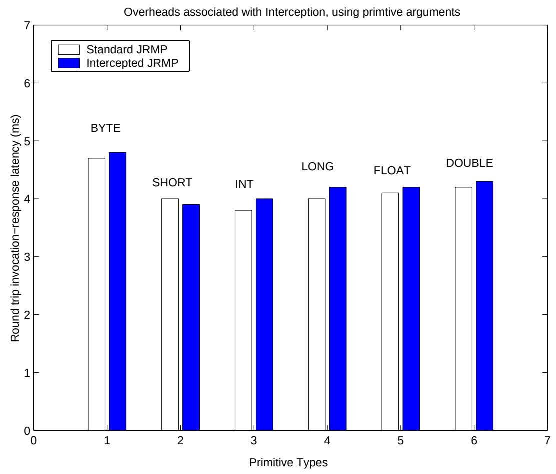

Figure 10. Latency introduced into RMI-JRMP applications by the SocketFactory Interceptor, for primitive arguments.

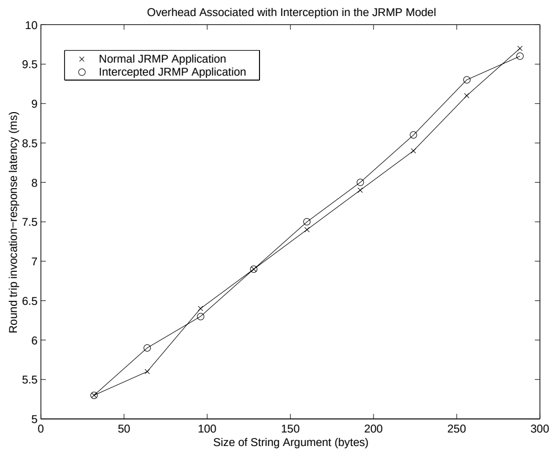

Figure 11. Latency introduced into RMI-JRMP applications by the SocketFactory Interceptor, for string arguments

into RMI-IIOP applications when appropriate. The experiments were conducted on a network of 167 MHz Sun Ultra-SPARC dual-processor machines running Solaris 2.6, and operating over a 100 Mbps Ethernet. Our performance measurement was the round-trip latency associated with a simple invocation-response, averaged over 100 round trips. The setup time – measured from the point when the client performs a lookup on the registry, to the point when it makes the first invocation – was also noted for reference, and represents a one-time cost. The test application consisted of a simple echo server deployed on the standard Sun Java Virtual Machine (version 1.2.2) for Solaris. The latency of a round-trip invocation and response was determined for parameters ranging from strings of varying length, to primitive types.

8.1 The Dynamic Proxy Approach

This interceptor exploits the Dynamic Proxy mechanism introduced in Java2 release 1.3, and makes use of the reflection capabilities of the Java model. Figure 8 and Figure 9 show the overheads associated with the Dynamic Proxy Approach to interception. The experiment was conducted with the SmartRegistry used in place of the standard *rmiregistry*. The interceptor was used to provide a simple logging service that displayed the parameters associated with the invocation. The overhead associated with just the interception mechanism varied between 1% to 15%, unlike in the other two approaches where the overhead remained almost constant. The average latency introduced by the Dynamic Proxy interceptor is 270 μs, with setup taking an additional 300ms when the interceptor is deployed. Although this interceptor has relatively poor performance, it is implemented in Java and is portable as a result. Deployment of the interceptor requires a SmartRegistry to be run in place of the *rmiregistry*; however, no modification of application code is required. In this respect, the interception is transparent to the application.

8.2 The RMISocketFactory Approach

The RMISocketFactory-based interceptor is also implemented completely in Java. In comparison to the other approaches, this interceptor registered the least overhead, and required just 16ms more for

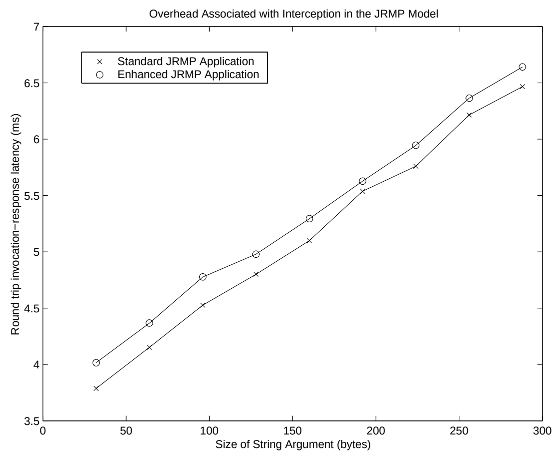

Figure 12. Latency introduced into RMI-JRMP applications by the Aroma Interceptor, for string arguments.

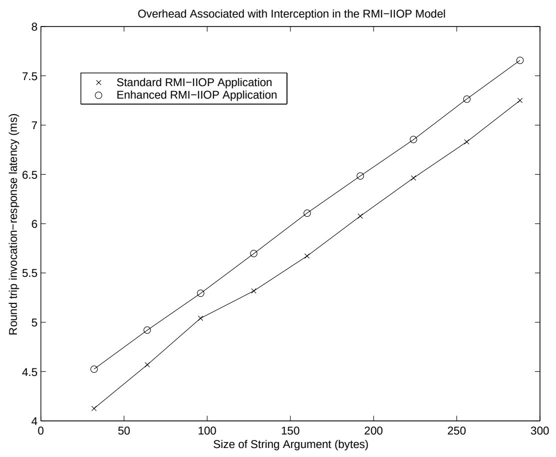

Figure 13. Latency introduced into RMI-IIOP applications by the Aroma Interceptor, for string arguments

setup when the interceptor was deployed. Figure 10 and Figure 11 show the overheads associated with exploiting the RMISocketFactory-based interceptor. The interceptor incurred a maximum overhead of 5.3%, and introduced an average latency of 100 μs to normal Java RMI operation. In addition to the low latency, this approach has the advantage of being portable, and of being supported by the Java RMI specification. However, analysis of invocation paramters requires the services of a parser component capable of understanding both JRMP and serialization formats; deploying such mechanisms adds appreciable overhead. Further, this interceptor cannot be deployed without modification of application code, requiring the addition of a single line of code to install the InterceptorSocketFactory.

8.3 The Library Mediation Approach

This interceptor exploits the Library Mediation approach described in Section 6. The measurements for this section were taken using the implementation of the interceptor used in the Aroma System. Thus, the overheads reflect the latency associated with deploying both the Aroma Interceptor and the Aroma Parser components. The Library Mediation approach is the only interceptor that provides a single solution applicable equally to both RMI-JRMP and RMI-IIOP applications.

Because the experiment did not involve the Message Handler and Multiplexer components of the Aroma System, we programmed the Parser to process the message and subsequently release the intercepted messages back into the default TCP/IP communication streams. Thus, the latency measured reflects the overhead associated with read-only interception. Our measurements were taken for four different configurations:

- A standard RMI-JRMP client with
  a standard RMI-JRMP server
- An Interceptor-enhanced RMI-JRMP client with
  an Interceptor-enhanced RMI-JRMP server
- A standard RMI-IIOP client with
  a standard RMI-IIOP server

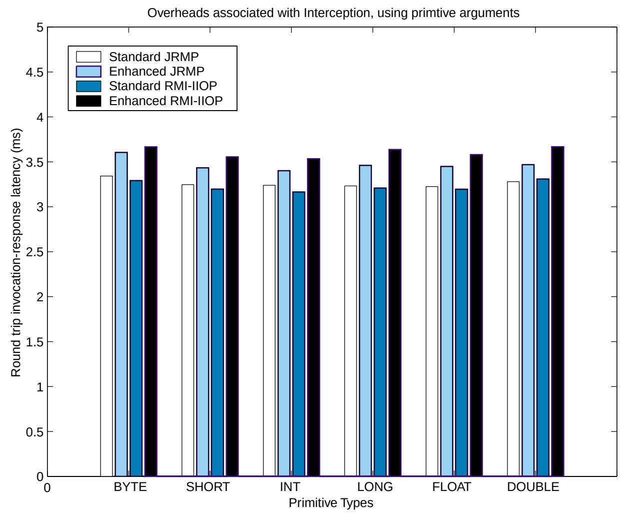

Figure 14: Latency introduced by the Aroma Interceptor for primitive arguments.

- An Interceptor-enhanced RMI-IIOP client with
  an Interceptor-enhanced RMI-IIOP server

In each case, we determined the latency of a round-trip invocation and response, for parameters ranging from strings of varying length, as shown in Figure 12 and Figure 13, to primitive types (`int`, `long`, `float`, etc.), as shown in Figure 14.

The results indicate that the Interceptor and Parser introduce an overhead of approximately 200 μs for RMI-JRMP applications, and 375 μs for RMI-IIOP applications. This difference can be attributed to the fact that RMI-IIOP messages use fixed-length headers that are considerably larger than those used by RMI-JRMP.

Latency overheads in the hundreds of microseconds are normally considered significant; however, it can be observed that the standard JRMI applications (both RMI-JRMP and RMI-IIOP) themselves exhibit a minimum latency of 3-4 ms. In this context, our Interceptor introduces less than a 10% overhead to the application performance. Moreover, as shown by the results, the performance of the JRMI applications deteriorates with increasing parameter size; regardless, our interception mechanisms incur an almost constant overhead.

9 Related Work

While interceptors are an accepted mechanism for CORBA [10], little work has been done on interception for the Java distributed object model. The Eternal System [8] exploits interceptors for providing transparent fault tolerance for CORBA applications. Eternal's Interceptor exploits the operating system's linker-loader facilities to interpose on networking libraries at the operating system level.

The primary development platform for the Interceptor has been Solaris. However, both Solaris and Linux provide additional facilities to support interception, such as the */proc* interface [1] for interception at the system call level. For the Windows NT operating system, the mediation connectors approach [2] defines mechanisms to build wrappers around dynamically linked libraries (DLLs) that can subsequently be used to mediate calls on those libraries; we can implement the Interceptor by exploiting these connectors to mediate calls to the `libnet` library.

10 Conclusion

Interceptors are software mechanisms that, when deployed, provide hooks to introduce new services to an application at runtime. We have developed three different interceptor mechanisms for applications that are based on the Java Remote Method Invocation model. These interceptors facilitate the capture of Java RMI messages, and can be exploited to analyze, modify or reroute these messages at runtime. The interceptors have been developed independent of the application, and are easy to deploy. Further, the interceptors can be deployed with minimal modification to the application. Our preliminary measurements show that, depending on the approach used, the overhead added by the interceptor is between 1% and 15%. With the SocketFactory interceptor, the maximum overhead measured was just 5%. The Dynamic Proxy approach showed relatively poor performance, but is easy to deploy, and can be used to provide simple fail-over capability for stateless server objects. With the Library Mediation approach, the overhead was less than 10%, but remained fairly constant for different kinds of applications and for different types of invocation parameters. This approach is applicable to both RMI-JRMP and RMI-IIOP applications, and is exploited in the Aroma System to enhance the existing Java RMI model with support for consistent replication of Java RMI client and objects.

References

[1] A. Alexandrov, M. Ibel, K. E. Schauser, and C. Scheiman. Ufo: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, 16(3):207–233, August 1998.

[2] R. Balzer and N. Goldman. Mediating connectors. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop*, pages 73–77, Austin, TX, June 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, July 1997.

[4] S. Maffeis. The object group design pattern. In *Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies*, pages 155–163, Toronto, Canada, June 1996.

[5] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[6] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Interception in the Aroma system. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 107–115, San Francisco, CA, June 2000.

[7] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Transparent consistent replication of Java RMI objects. In *Proceedings of the Distributed Objects and Applications Conference*, Antwerp, Belgium, September 2000.

[8] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance CORBA. *IEEE Computer*, pages 62–68, July 1999.

[9] Object Management Group. *The Common Object Request Broker Architecture and Specification*, 1998. Version 2.3 OMG Technical Committee Document (formal/98-12-01).

[10] Object Management Group. *Portable interceptors, Revised Joint Submission*, December 1999. OMG Technical Committee Document (orbos/99-12-02).

[11] Sun Microsystems, Inc. *Java Remote Method Invocation Specification*, revision 1.50 edition, October 1998. http://java.sun.com/products/jdk/rmi/index.html.

[12] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996. MIT Press.