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

An interceptor is a software mechanism that provides the hooks that are needed to introduce additional code dynamically into the execution path of an application. By exploiting interceptors, developers can enhance and potentially modify the behavior of an application at runtime without having to revise or recompile the application code. We have identified three distinct interception points for the Java Remote Method Invocation model, at the proxy level, the transport level and the shared library level of the JavaRMI model, respectively. The interceptors implemented at these interception points employ the DynamicProxy API, the RMISocketFactory API, and a library mediation approach, respectively. Our interceptor implementations are novel in that they are transparent to the application, add nominal latency overheads, and are easy to deploy, requiring only minimal modifications to the application. We describe how the interceptors can be exploited to introduce additional services (such as logging and profiling mechanisms) to the JavaRMI runtime. In particular, we describe the use of interceptors in the Aroma System to enhance the existing JavaRMI model with support for fault-tolerance through the consistent replication of JavaRMI objects.

1 Introduction

The Java Remote Method Invocation (JavaRMI) model [11, 12] simplifies the design and development of distributed Java applications by making the complexities of network programming transparent to the application programmer. JavaRMI supports access transparency (clients invoke methods on remote servers exactly as though the servers were local) and location transparency (clients obtain access to remote servers without having prior knowledge of the server address); however, it does not provide mechanisms for system monitoring, profiling, message logging, security or reliability. If such services are desired, they must be developed independently, and integrated into every application that requires them.

Interceptors provide an alternative mechanism for adding new services to an application at runtime, in a transparent manner. An interceptor is a software component that provides a hook into the execution path of an application at select "interception points". Programmers can exploit these hooks to add custom code that is executed at runtime, at that interception point, in a transparent manner. For instance, a socket-level interception point could be used to introduce code that monitors the frequency and destination of invocations generated by a local JavaRMI client.

Through its support for "portable interceptors" for CORBA [9, 10], the Object Management Group has recognized the need for interception as a means for adding new services to CORBA applications in

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

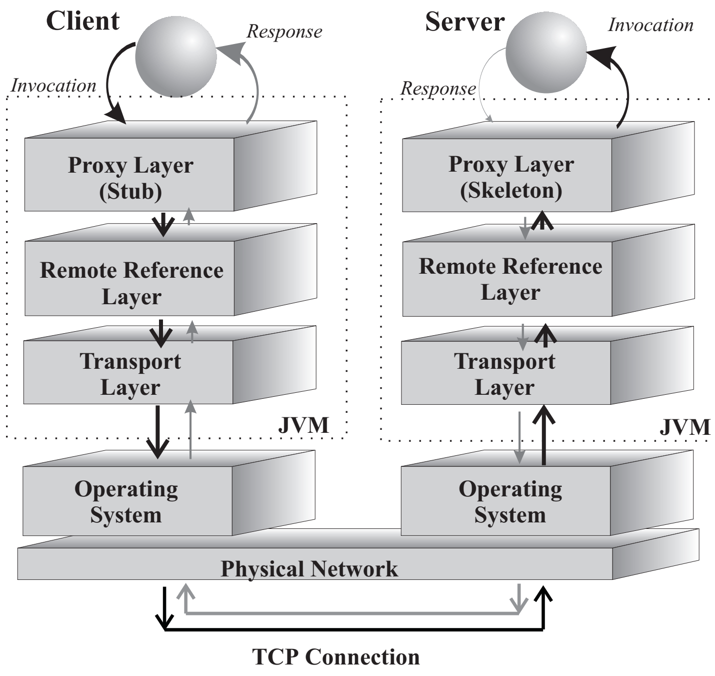

Figure 1: The Java Remote Method Invocation Model.

a transparent, flexible and portable manner. However, the JavaRMI model does not currently provide any comparable mechanism. We address this deficiency by identifying three distinct interception points within the JavaRMI infrastructure – at the proxy level, the transport level and the shared library level respectively – and by deploying interceptors at these points.

The interceptors are novel in that they are transparent to the application, easy to deploy, and add a nominal overhead to the latency associated with JavaRMI operations. Depending on the location of the interception point, the interceptors can be exploited to provide services that include message logging, message routing, system profiling and monitoring. In this paper we describe the use of interceptors in enhancing the reliability and availability of a distributed Java application. In particular we exploit interception in the Aroma System that we have developed, to enhance the existing JavaRMI model with support for fault-tolerance through the consistent replication of JavaRMI objects.

2 The JavaRMI Architecture

Because our interceptors are targeted at the Java Remote Method Invocation model, it is necessary to understand the JavaRMI architecture to identify potential interception points. Figure 1 shows the JavaRMI protocol stack consisting of the transport layer (to setup and manage TCP/IP connections), the reference layer (to validate the semantics of the JavaRMI operation) and the proxy layer that provides the application interface to the JavaRMI infrastructure.

A JavaRMI 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 JavaRMI 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 a JavaRMI server must first obtain a stub for that server. This bootstrapping process is facilitated by the *rmiregistry*, a nameserver that maintains a transient database of server stubs mapped against the stringified names of the corresponding servers. Active JavaRMI servers `bind` their stubs against well-known names in the rmiregistry; a client equipped with a server's name can `lookup` the registry to retrieve that server's stub. Once bootstrapping is completed, the registry plays no further role in the client-server communication.

A JavaRMI operation is initiated by the client making an invocation on the locally 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 JavaRMI 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 typically a non-application software component that provides hooks to introduce custom code at select points in the execution path of the application, at runtime. Our primary objective is to develop interceptors that can be exploited to introduce replication mechanisms to the JavaRMI model to improve the reliability and availability of distributed Java applications.

Replication involves the distribution of multiple copies of an object across different processors in the system. Consequently, we require mechanisms to provide one-to-many communication, and to ensure that the replicas maintain a consistent state. For instance, with an actively replicated server, every client invocation must be transmitted to all replicas of the server; although multiple responses may be generated as a result, only a single response should be returned to the client.

To facilitate replication in a manner that is transparent to the application, we require interceptors that can capture the unicast JavaRMI messages (invocations and requests) before they reach the physical network, and can divert these messages to the mechanisms required for replication. These mechanisms typically provide services that adapt the TCP/IP message for multicast to all replicas of the destination server object, maintain replica consistency, perform message logging and fault detection, and supervise the recovery of a failed replica. While the interceptor in this scenario is primarily responsible for message capture and re-routing, it can also embed code to analyze, modify or enhance the contents of the intercepted messages. Therefore, we define three modes of operation for the interceptor:

- *Read-only* mode where the message contents are left unchanged.
- *Enhancement* mode where the message contents are augmented.
- *Transformation* mode where the message contents are modified.

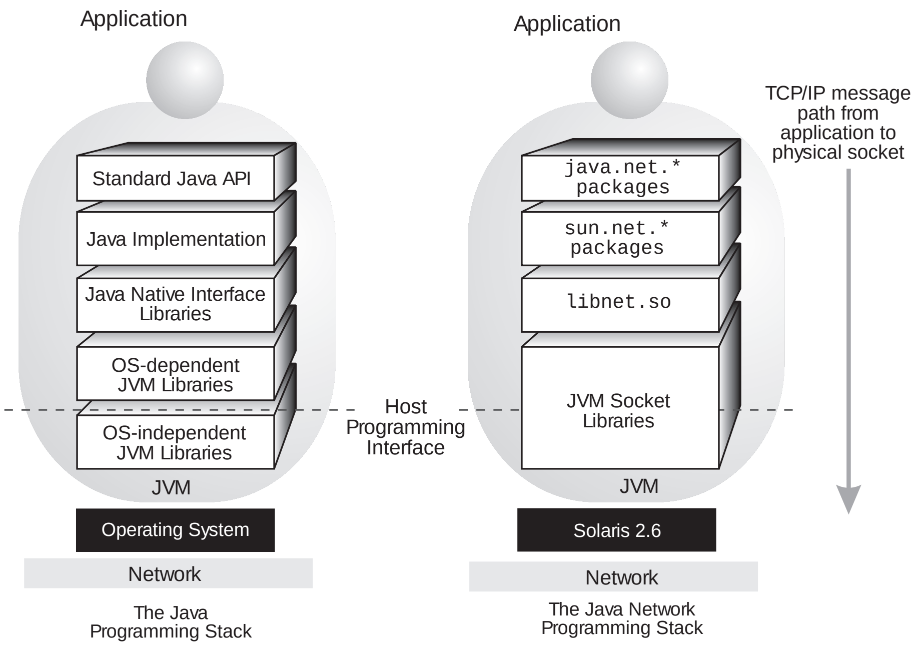
Application
Standard Java API
Java Implementation
Java Native Interface Libraries
OS-dependent JVM Libraries
OS-independent JVM Libraries
JVM
Operating System
Network
The Java Programming Stack
Host Programming Interface
Application
java.net.* packages
sun.net.* packages
libnet.so
JVM Socket Libraries
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.

Depending on the mode of operation, the interceptor can provide different levels of service. For instance, a read-only interceptor can be extended to provide a message logging service, while an enhancement interceptor can be used to append contextual information to messages, and a transformation interceptor can be used to encrypt the messages for secure communication. Because the enhancement and transformation interceptors have the potential to introduce behavior that is visible to the application (an undesirable consequence), we enforce the following rule when deploying interceptors. For every enhancement and/or transformation interceptor in the communication path, there must exist a complementary interceptor further along the message path that removes the enhancement and/or reverses the modification, before the message is delivered to the application.

3.1 Interception Points in the JavaRMI model

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

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 be required to handle some portability issues.

- The `Java Virtual Machine Socket Libraries` level. This level hosts the native libraries for the Java Virtual Machine (JVM); the libraries contain code that is either independent of the operating system, or that maps onto operating system-specific functionality. Thus, interception at (or below) this level will involve serious portability problems.

The decision to define an interception point at any of these levels is based on three factors. First, there must exist at the level, some function or property (i.e. a "hook") that we can exploit to deploy the interceptor at runtime. Second, the performance penalty incurred at runtime (due to the execution of additional code at that level) must be acceptable. Third, the portability of both the interceptor code and any custom code introduced at that level must be taken into account.

Based on the first of the above factors, we have identified three interception points for the Java Remote Method Invocation model. The first option places the interception point at the proxy layer of the JavaRMI protocol stack, and exploits the functionality provided by the Dynamic Proxy API. The second option employs interception at the `java.net` level of the networking stack by exploiting the `RMISocketFactory` API. The final option places the interception point at the Java Native Interface (JNI) level of the networking stack by exploiting a mechanism (described in Section 6) that we refer to as library mediation. When choosing an interceptor mechanism to provide a specific service, we can use performance and portability as the criteria to evaluate the relative merits of each of the above options.

4 The Dynamic Proxy Approach

The Dynamic Proxy API is a recent addition to the Java2 Standard Edition. It exploits the reflection capabilities of the Java platform to create, at runtime, a proxy for any object that implements a specified list of interfaces. A method invocation on any one of the supported interfaces is encoded and dispatched by the proxy, through a uniform interface, to an object that implements that method. Because the proxy always deals with this uniform "handler" interface, the details of the implementation are completely transparent to it. Consequently, a single proxy could potentially represent one or more backend objects that collectively implement the interfaces supported by the proxy. Thus, the dynamic proxy effectively isolates the invocation interface from the implementation of that interface, using a handler object to mediate all interactions between them.

Figure 3 shows the implementation details of the Dynamic Proxy API. The key entities are the `java.lang.reflect.Proxy` class and the `java.lang.reflect.InvocationHandler`, as shown in Figure 3. When a method invocation occurs on an instance of the Dynamic Proxy class, the parameters of the invocation are encoded, using reflection, into a `java.lang.reflect.Method` object and a `java.lang.Object[]` array 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

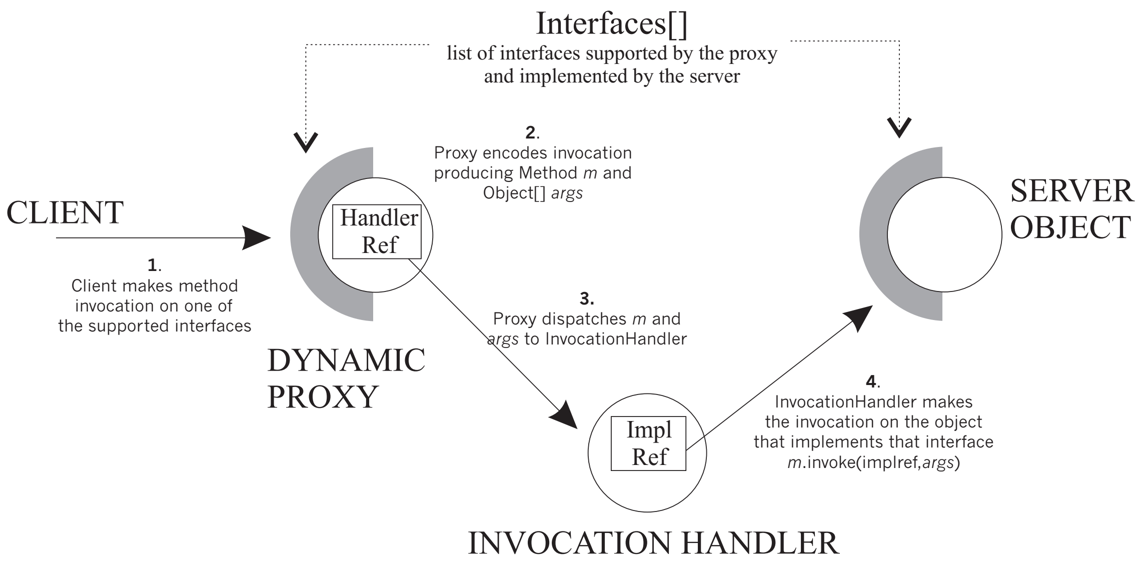
Interfaces[]
list of interfaces supported by the proxy and implemented by the server
2. Proxy encodes invocation producing Method m and Object[] args
CLIENT
Handler Ref
SERVER OBJECT
1. Client makes method invocation on one of the supported interfaces
3. Proxy dispatches m and args to InvocationHandler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

Figure 3: The Dynamic Proxy mechanism.

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 maintain references to one or more objects that collectively implement those interfaces. In the former case, the delegate processes the invocation directly, while in the latter case, the delegate in turn invokes the method on the appropriate implementation.

4.1 Design

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 JavaRMI server, and creating a custom `InvocationHandler`, we can intercept all invocations destined for that server.

To achieve this, the set of interfaces implemented by the Dynamic Proxy must correspond to the set of `Remote` interfaces implemented by the JavaRMI 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 JavaRMI server reference, and consists of a TCP/IP endpoint and an object identifier. With the existing JavaRMI APIs, there is no direct method to obtain the actual server reference; 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. However, every invocation now has additional levels of indirection i.e., the invocation travels from client to proxy, from proxy to invocation handler, from invocation handler to stub, and from stub to server before the result can be returned.

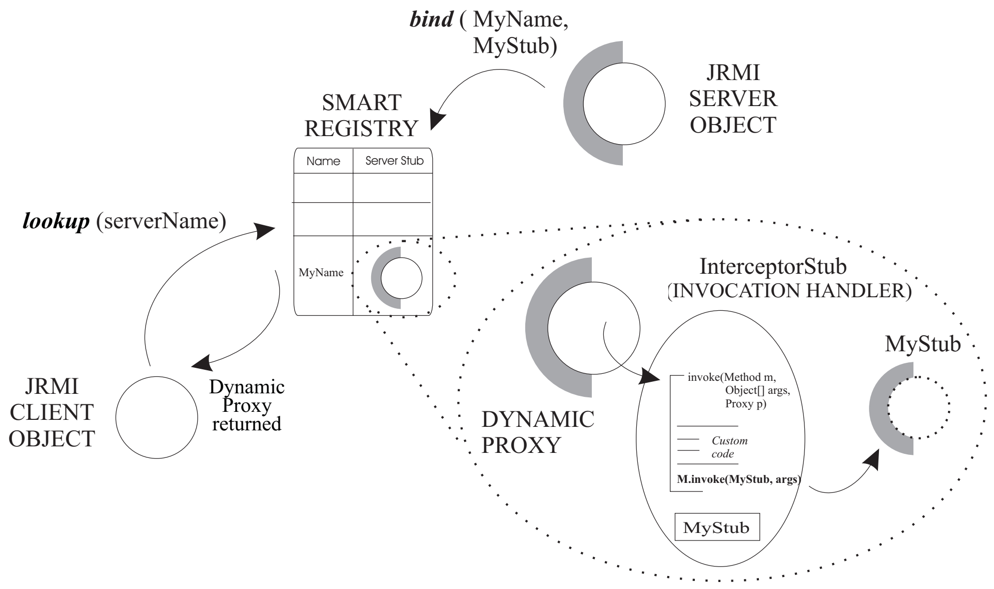

Figure 4: The Dynamic Proxy approach to interception.

4.2 Implementation

Our simple interceptor mechanism consists of a Dynamic Proxy generated for the JavaRMI 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 JavaRMI 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 `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 example shown in Figure 4, the custom code logs the invocation parameters before delegating the invocation. In Section 7.1, we describe a simple failover example where we maintain multiple references (corresponding to the replicas of a server) at the invocation handler, and insert custom code to detect the crash of a server replica. The intent is to failover to an alternative replica and retry the invocation thus providing fault transparency to the client.

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 the application. Our solution is to use a custom SmartRegistry. The SmartRegistry subclasses the default registry implementation and provides a drop-in replacement for the standard *rmiregistry*, as illustrated in Figure 4. Thus, it can be invoked in exactly the same manner as the standard *rmiregistry*, with no modification to the client or server.

The SmartRegistry maintains a transient database that maps a server name onto a Dynamic Proxy for that server, rather than onto the server stub. When a server invokes the `bind` method on the SmartRegistry, it passes its standard stub and a stringified server name as parameters to the bind method. 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

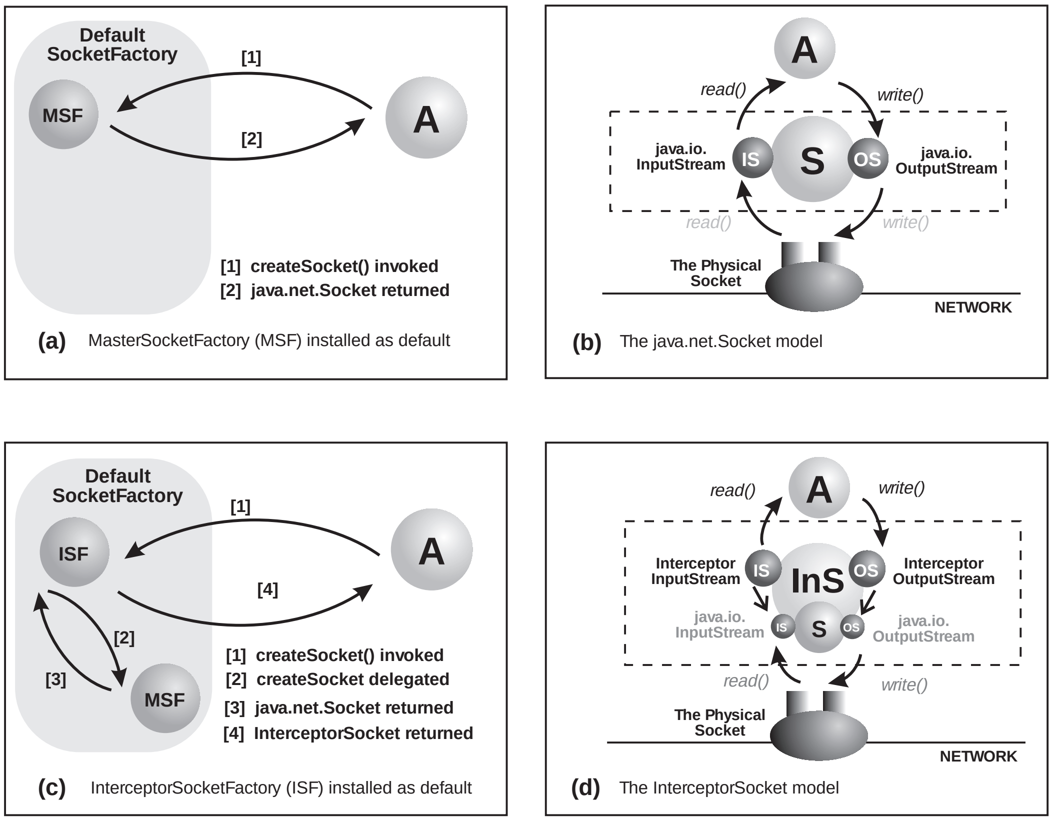
Default SocketFactory
[1]
MSF
[2]
A
[1] createSocket() invoked
[2] java.net.Socket returned
(a) MasterSocketFactory (MSF) installed as default
A
read()
write()
java.io. InputStream
IS
S
OS
java.io. OutputStream
read()
write()
The Physical Socket
NETWORK
(b) The java.net.Socket model
Default SocketFactory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

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 JavaRMI 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.

We exploit this functionality to implement the interceptor. Our interceptor package contains 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 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 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 necessarily pass through these objects, we can place custom code within them to implement our Interceptor. The socket factory interceptor is inherently portable, being implemented completely in Java. While the interception mechanisms are transparent to the application, the deployment of the interceptor 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. Thus, although this approach involves some modification to the application code, the changes are minor.

In RMI-JRMP, a custom RMISocketFactory once deployed becomes an integral part of the server reference; this ensures that both client-side and server-side sockets use compatible socket factories to communicate. However, because RMI-IIOP objects are intended for interaction with CORBA-compliant objects, we could potentially have a remote non-Java object that is incapable of exploiting the RMISocketFactory for socket creation. Thus, the RMISocketFactory mechanism is not supported for RMI-IIOP objects.

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. The Library Mediation approach introduces the interceptor at a lower level in the implementation hierarchy, and ensures that the interceptor is activated in a manner that is transparent to the application.

In the library mediation approach, we take advantage of the fact that the Java Virtual Machine (JVM) uses the services of a Java "library loader" to load all native libraries requested 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. By implementing a custom version of any native library, and pre-pending its location to `sun.boot.library.path`, we can "trick" the loader into loading our custom library in place of the standard native library. Further,

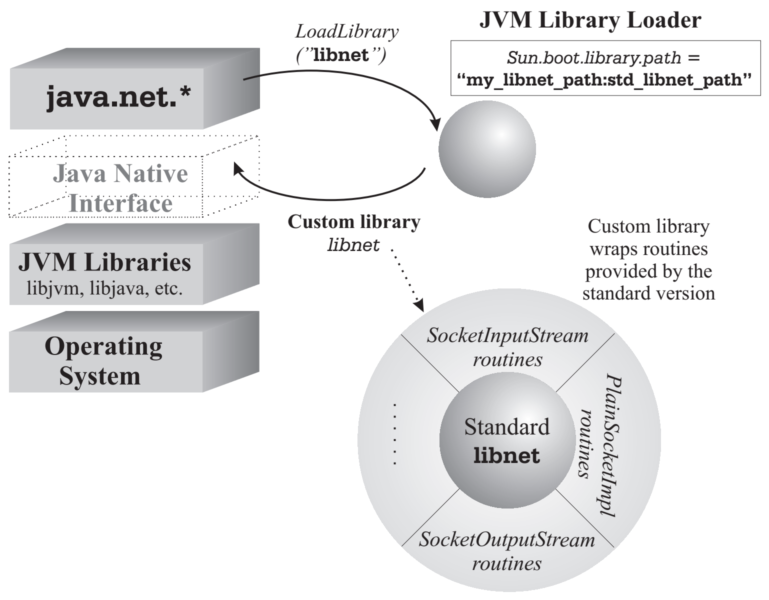

Figure 6: The Library Mediation Approach.

if we cache a handle to the standard library within the custom library, we can effectively preprocess calls in the custom library before delegating them to their standard implementations. We refer to this approach as *library mediation*.

The Java networking model employs two sets of native libraries, one at the Java Native Interface (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 requires 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 Java distribution provides a `javah` utility that extracts from a Java class, a native header file containing the signatures of all native (JNI) 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. In this paper, we use *myLibnet* to refer to our custom implementation of the `libnet` library.

The *myLibnet* library provides the ideal location to embed our interceptor code. We develop *myLibnet* as a collection of wrappers around the routines of the standard `libnet` implementation; by default, all calls on the *myLibnet* 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 *myLibnet* 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 for Reliability

Our primary objective in investigating interceptors for Java Remote Method Invocation is to use an interceptor, in the Aroma System, to introduce services to ensure reliable, highly available system operation. Object replication is a well-known approach to achieving the latter objectives, requiring mechanisms to distribute replicas across the system and to ensure that they maintain a consistent state. Depending on the type of object being replicated (stateless vs. stateful) and the desired functionality (fault-tolerance vs. high availability), we can employ different replication schemes. In the following sections, we look at two different applications of interceptors in this context. First, we use the Dynamic Proxy interceptor to implement a simple failover mechanism for stateless servers Next, we exploit the Library Mediation interceptor in the Aroma System to transparently enhance the existing JavaRMI model with support for fault-tolerance through the consistent replication of JavaRMI objects.

7.1 Simple Failover using the Dynamic Proxy Approach

Stateless JavaRMI server objects can be replicated either for fault-tolerance or for high 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. If the objects are replicated for availability, we need mechanisms that can route distinct client requests to different server replicas based on the current load at each replica. If the objects are replicated for fault-tolerance, application transparency to faults can be achieved by using a simple failover mechanism. An invocation is forwarded to any one of the available replicas; in the event that the replica fails, the mechanisms detect the fault, select an alternative server replica and retry the invocation, thus masking 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.

Replication of stateless servers, for high availability or for failover, can be achieved by exploiting the Dynamic Proxy interceptor. In this case, the SmartRegistry implementation is modified to allow multiple servers to bind themselves to the same name. An 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 looks up the SmartRegistry will retrieve a Dynamic Proxy and its associated InvocationHandler. The failover mechanisms is implemented by 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 invalidates the first stub, retrieves a second stub from its local array, retries the invocation, and returns the result thereby masking the fault from the client. The mechanism can tolerate up to *n-1* faults, where n is the initial number of "live" stubs cached at the InterceptorStub. The mechanism can also be extended to support recovered servers. A recovered server that binds itself in the SmartRegistry will overwrite its old reference (stub) with the new live reference. When the InterceptorStub has exhausted all its cached stubs, it can be programmed to contact the registry and replenish its cache with stubs to new/recovered replicas, thereby continuing to provide failover capability to the application.

7.2 Consistent Replication Using the Aroma System

The dynamic proxy provides a simple, portable mechanism for implementing failover for stateless JavaRMI servers. 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 JavaRMI 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 detailed information can be found in [6, 7].

The Aroma System adopts the object group paradigm [4] for transparent replication of JavaRMI 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 within and between 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 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 *read*s, the Interceptor can manipulate inbound messages to the application; similarly, by handling the *write*s, it can control the format and content of outbound messages generated by the application. Because Aroma intercepts *every* TCP/IP call made by the ap-

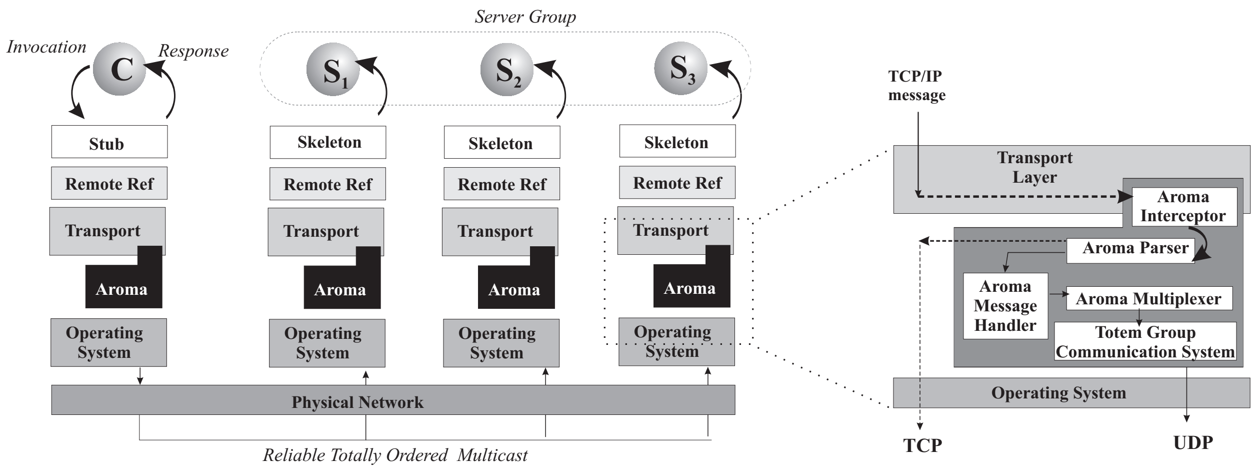

Figure 7: The Aroma System Architecture.

plication, the Aroma Parser is needed to filter those TCP/IP messages that conform to known JavaRMI formats. All valid JavaRMI messages are forwarded to the Aroma Message Handler; all other TCP/IP messages are released and allowed to 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 JavaRMI 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 JavaRMI, 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 deploying the three different interceptor implementations. The experiments were conducted on a network of 200MHz Sun Ultra-SPARC single processor machines, running Solaris 2.7, and operating over a 100Mbps Ethernet. All measurements were taken using the Java2 Standard Edition (build 1.3.0) with the Java HotSpot Client VM (build 1.3.0). The measured quantity was the round-trip latency associated with a simple invocation-response, averaged over 1000 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

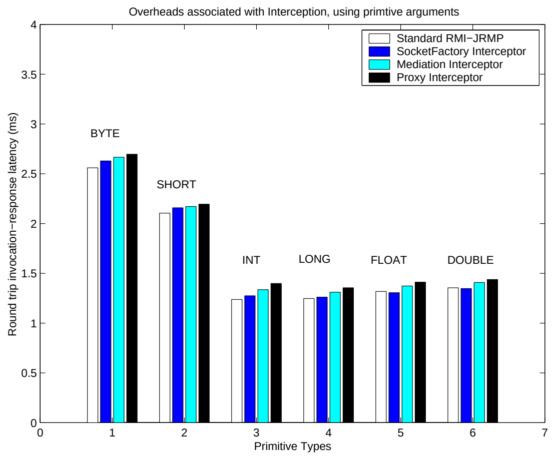

Figure 8. Latency measurements for RMI-JRMP applications using the three interceptor implementations, for primitive arguments.

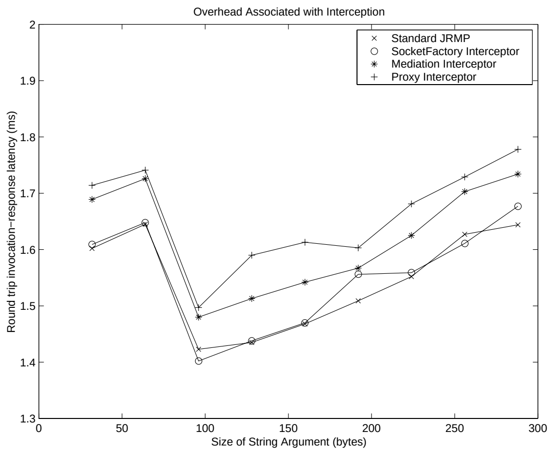

Figure 9. Latency measurements for RMI-JRMP applications using the three interceptor implementations, for string arguments.

cost. The test application consisted of a simple echo server, with latency measurements determined for parameters ranging from primitive data types to strings of varying length.

Figure 8 and Figure 9 show the overheads of the SocketFactory Interceptor, the Library Mediation interceptor and the Dynamic Proxy interceptor with reference to the latency associated with the unintercepted RMI-JRMP application. Figure 10 and Figure 11 show the overheads associated with the Library Mediation interceptor, both for RMI-JRMP applications, and for RMI-IIOP applications.

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. For the Dynamic Proxy measurement, the experiment was conducted with the SmartRegistry instead of the standard *rmiregistry*. The overhead associated with the interception mechanism varied between 74 μs and 161 μs. The average latency introduced by this interceptor is 115 μs. This translates to an overhead of between 5% and 11%, the highest overhead recorded among the three interceptors. The setup time for dynamic proxies is also the highest among the interceptors, requiring an additional 200 μs.

This setup cost can be attributed to the fact that the "stub" downloaded at setup contains additional classes (i.e. the dynamic proxy and the invocation handler classes) that need to be deserialized. The high latency was anticipated not only because of the high costs associated with reflection, but also because every invocation passes through at least three entities (proxy, handler and stub) before it is delivered to the remote endpoint. 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 smallest overheads (between 0.1% and 3%), and

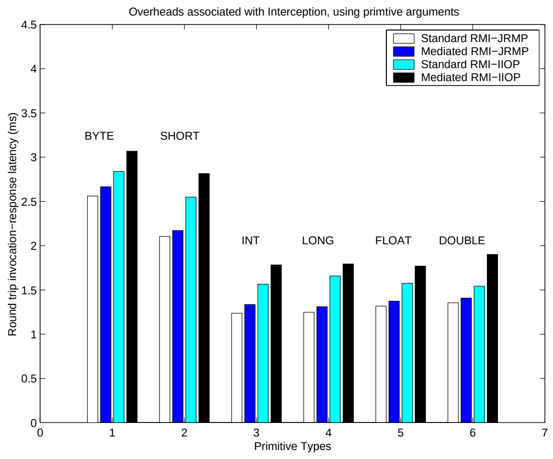

Figure 10. Latency introduced by the Library Mediation Interceptor for primitive arguments, for RMI-JRMP and RMI-IIOP applications.

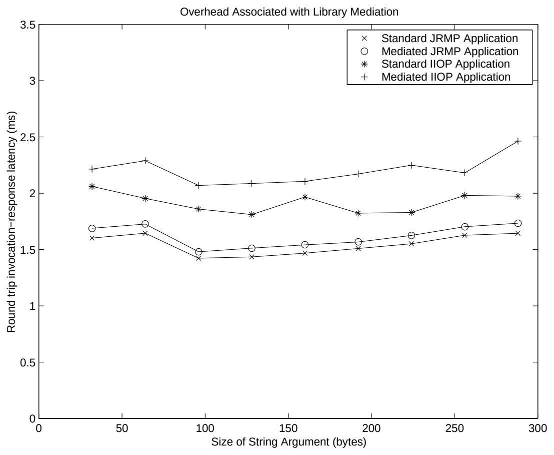

Figure 11. Latency introduced by the Library Mediation Interceptor for string arguments of varying length, for RMI-JRMP and RMI-IIOP applications.

required almost no additional setup time compared to the reference implementation. Figure 8 and Figure 9 show the overheads associated with exploiting the RMISocketFactory-based interceptor.

The interceptor incurred a maximum overhead of 69 μs (for primitive byte arguments), and introduced an average latency of 7 μs to normal JavaRMI operation. In addition to the low latency, this approach has the advantage of being portable, and of being supported by the JavaRMI specification. However, analysis of invocation parameters 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 the 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. However, the interceptor was modified to eliminate the parsing component. Thus, the latency measured reflects the overhead associated with read-only interception. The Library Mediation approach is the only interceptor that provides a single solution applicable equally to both RMI-JRMP and RMI-IIOP applications. 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
- 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 primitive types (`int`, `long`, `float`, etc.), to strings of varying length, as shown in Figure 10 and Figure 11.

The results indicate that the Interceptor introduces an overhead of approximately 71 μs for RMI-JRMP applications, and 285 μs for RMI-IIOP applications. The RMI-IIOP messages typically use fixed-length headers that are larger than the standard RMI-JRMP message header. The RMI-IIOP implementation also generates more calls to the networking library; for instance, a *read* requires one call to read in the GIOP header to determine the message length, followed by a second call to read in the actual message. Because each call to the library is intercepted, the RMI-IIOP application has a relatively larger interception overhead compared to RMI-JRMP.

Latency overheads in the hundreds of microseconds are normally considered significant; however, for standard RMI-JRMP applications, the latency is usually in the order of a couple of milliseconds. Thus, the Library Mediation interceptor adds an overhead of 4%-5% to RMI-JRMP applications. Moreover, as shown in Figure 11, while the performance of RMI-JRMP applications deteriorates with increasing parameter size, 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 to 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, in a transparent manner, with minimal modifications to the application. 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 JavaRMI 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 with minimal modification to the application. Preliminary measurements show that, depending on the approach used, the overhead added by the interceptor is between 0.1% and 11% for RMI-JRMP applications.

The SocketFactory approach introduces the least overhead for interception, and is born portable. However, because the interception occurs at such a high level in the protocol stack, it limits the functionality accessible to, and modifiable by, the interceptor. In the context of replication, a primary limitation of this interceptor is its tight coupling to the TCP/IP transport model; mapping intercepted messages to alternative transport protocols at this level, is a complex and expensive undertaking. The Dynamic Proxy approach is the most expensive of the interceptor approaches. While it displayed relatively poor performance, it is easy to deploy, is born portable, and can be exploited to provide transparent failover capability for stateless JavaRMI servers. The Library Mediation approach exhibited a higher overhead than the SocketFactory interceptor; however, the overhead is an acceptable 4%

- 5%, and remained fairly constant with varying parameter types and lengths, for RMI-JRMP applications. It should be noted that the LibraryMediation interceptor provides access to a wider range of functions (i.e., the entire networking library) at a much lower level. Thus, we exploit the Library-Mediation approach in the Aroma System to adapt the JavaRMI model for multicast communication, and to enhance the existing JavaRMI model with support for consistent replication of JavaRMI 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*, pages 17–26, 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 2.4 OMG Technical Committee Document (formal/2000-10-01).

[10] Object Management Group. *Portable interceptors, Revised Joint Submission*, December 1999. OMG Technical Committee Document (ptc/2000-03-03).

[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.