Interceptors for Java™ Remote Method Invocation

Weiliang Li (Lehigh University) and Dale Parson (Agere Systems)

Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications

Volume II, p. 850-856.

Computer Science Research, Education, and Applications Technology Press

Las Vegas, Nevada, June 25-28, 2001.

Interceptors for Java™ Remote Method Invocation

Weiliang Li
wel3@eecs.lehigh.edu
Lehigh University, EECS Dept.
19 Memorial Drive West
Bethlehem, Pa. 18015

Dale Parson
dparson@agere.com
Agere Systems
1247 South Cedar Crest Blvd.
Allentown, Pa. 18103

Keywords

interceptor, CORBA, Java, remote procedure call, remote method invocation

Abstract

A software interceptor is a procedure or object that interposes itself between an invoking client and an invoked server software entity. Procedural interceptors can redirect procedure invocations to alternative procedures that are selected dynamically at run time. Procedural interceptors are useful for program profiling, tracing, dynamic delegation of operation implementation, and interactive debugging. Distributed interceptors for remote procedure calls support additional capabilities such as protocol bridges, server thread schedulers, fault tolerant replication, security audits and encryption, and performance enhancements such as compression and caching. CORBA, the Common Object Request Broker Architecture, comes with standard portable interceptors, but interceptors are missing from Java Remote Method Invocation (RMI). Java reflection assists adding custom interceptors to RMI by enabling a generator to create interceptor classes automatically. These classes interpose on both the client and server sides of RMI invocations, supporting interceptor-based extensions that are transparent to client code.

1. Introduction to interceptors

A software *interceptor* is a procedure or object that interposes itself between an invoking client and an invoked server software entity, e.g., between a calling and a called procedure. A client invokes an interceptor using the same operation names and parameter types and values that it would use to invoke a server directly. Indeed a client is usually unaware that it is invoking an interceptor.

Once invoked, an interceptor performs some analysis or transformation on input parameters. An interceptor may then invoke the server operation of the same name, using the original parameters or transformed parameters. When the server operation returns, the interceptor may analyze or transform its results before returning them to the invoking client. Alternatively, an interceptor may bypass the intended server operation altogether, performing its own computations and returning results to the client.

A key feature of interceptors is *dynamic interposition*. An interceptor introduces a new layer into a layered software system by interposing itself between clients and servers during program execution. Transparent, dynamic interposition requires support from the run-time system that executes a target program. The run-time system provides some means for an interceptor to intercept invocations that are en route from clients to servers.

In Listing 1 we give a simple example of procedural interception using the interpreted Tcl scripting language [1]. We start out by exercising a procedure *avglist* that computes the mean of a list of input numbers. After determining that this procedure works as expected, we interpose an interceptor by moving avglist into namespace *hidden*, and then defining a new procedure *avglist*

— the interceptor — that records the maximum input number passed from any client, after which the interceptor invokes the original procedure and returns its result. Once the interceptor is installed, we can check global variable *max* to determine the maximum number passed. An alternative interceptor might bypass the original avglist, returning the median or mode instead of the mean. Client code that invokes avglist remains unaware that interception is occurring.

```
proc avglist {numbers} {# average of
    set sum 0.0             ; # numbers
    set count 0
    foreach entry $numbers {
        set sum [expr $sum + $entry]
        incr count }
    return [expr $sum / $count] }
% avglist {2 4 6 8}
5.0
% # Now interpose an interceptor.
rename avglist hidden::avglist
proc avglist {numbers} {
    global max
    foreach entry $numbers {
        if {$entry > $max} {
            set max $entry }}
    return [hidden::avglist $numbers] }
% set max -1000 ; # initialize
-1000
% avglist {2 4 6 8}
5.0
% echo $max ; # check intercepted max
8
```

Listing 1: A simple Tcl interceptor

Procedural interception in an interpreted language like Tcl is possible because the program interpreter resolves procedure name-to-code bindings at run time. An interceptor can alter a binding by renaming a procedure, by moving it to a private namespace, or by using some other language-specific mechanism. The interceptor creates its own name-to-code binding in place of the original. There can be multiple levels of interceptors for a single server procedure, where each interceptor remains unaware of the existence of other interceptors. The *rename* command in Listing 1 may in fact be renaming an interceptor. As long as each interceptor uses a different namespace to hide its "original" command, interceptors can nest to arbitrary depth.

Run-time environments for compiled programs use other mechanisms to support procedural interception. Some compiled programs are linked so they can attach to *dynamic link libraries* at program initialization time. A run-time loader binds unresolved procedure references in such programs to compiled procedures by traversing a library path in search of libraries with appropriate names and symbols. In such environments an interceptor can interpose itself into invocations of library procedures by interposing an interceptor library into the search path [2]. A client invocation of a dynamically loaded procedure enters the interceptor, and the interceptor can call the original library explicitly.

Another interception mechanism for compiled programs uses process control and trace system calls such as /proc and ptrace() for UNIX [3]. An interceptor acts as a special-purpose debugger that places breakpoints in application procedures and system calls. When a target program triggers an interceptor's breakpoint, the interceptor performs its work. The interceptor can invoke or bypass the intercepted procedure.

Procedural interceptors enable a number of useful dynamic processing capabilities [2,4].

(i) They enable *profiling*. An interceptor can count invocations, measure time spent in invocations, measure quantity of data passed in parameters, and measure application-specific properties of parameters and return values.

(ii) They enable *tracing*. Interceptors can log nested invocations of server procedures, thereby enabling *trace-based debugging* of client-server interactions.

(iii) They enable *dynamic delegation* of operation implementation. A chain of one or more interceptors embodies the *Chain of Responsibility* design pattern [5]. This pattern is appropriate when there may be more than one handler for an operation, when decoupling between an operation client and server is desired, or when late binding of a servant to an operation is desired. Any operation that allows a user or framework to specify an exact operation at run time is a potential candidate for this use of interceptors.

(iv) They enable *interactive debugging*. An interceptor can invoke an interactive debugger before or after invoking a server's operation. The debugger can inspect and possibly modify parameters, returns values, and other elements of program state.

2. RPC and CORBA interceptors

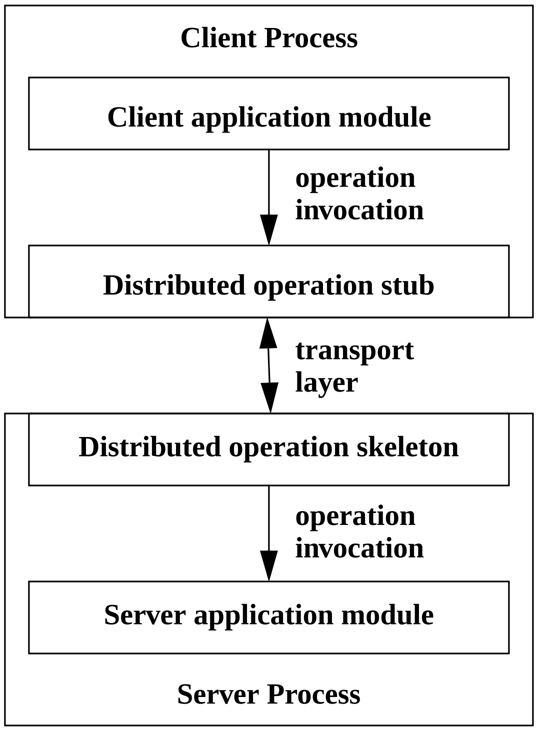
While procedural interceptors have several uses as discussed in Section 1, they become even more useful when applied in Remote Procedure Call (RPC) systems such as the *Common Object Request Broker Architecture* (a.k.a. *CORBA*) [6,7]. Figure 1 gives an outline of distributed procedure invocation processing in an RPC system. A client invokes an operation on a stub module contained within the same networked node and process as the client. The stub marshals parameters into a data stream according to the protocol requirements of its transport layer, then ships this data stream to the server process via this transport. A server-side skeleton demarshals the data stream back into parameters, then it invokes the corresponding server operation. When the operation completes, these steps are reversed. The server skeleton marshals return values and ships them back to the client stub via the transport, and the stub demarshals them and returns them to the client. CORBA expands on this basic RPC mechanism by supporting communication among clients and servers built using differing programming languages and running on differing computer architectures. CORBA also offers an assortment of distributed processing services.

RPC increases the number of potential interception points for an operation from two to four. Whereas a conventional, single-process procedure can be intercepted both before and after invocation, a remote procedure can be intercepted in the client process before invocation, in the server process before and after invocation, and again in the client process after invocation.

The presence of the network transport layer adds several new uses for interception that are not present in single-process applications [4].

(i) RPC interceptors can serve as *bridges* to alternate communications protocols. An interceptor can remarshal data streams to conform to protocols not envisioned by the original distributed application designers.

(ii) RPC interceptors can serve as *schedulers* by overriding default thread allocation policies in multi-threaded servers.



Client Process
Client application module
operation invocation
Distributed operation stub
transport layer
Distributed operation skeleton
operation invocation
Server application module
Server Process

Figure 1: Remote procedure invocation

(iii) RPC interceptors can support *fault-tolerant processing* by *replication* of redundant servers and *auditing* of replicated invocations.

(iv) RPC interceptors can support distributed processing *security*, for example *authentication* of clients, as well as pre-transport *encryption* and post-transport *decryption* of data streams.

(v) RPC interceptors can support performance enhancement for distributed processing, such as pre-transport *compression* and post-transport *decompression* of data streams, as well as *caching* of stateless distributed query results in a local cache server.

Early versions of CORBA supported application-specific or vendor-specific types of interceptors [4,8]. Current CORBA includes a vendor-neutral standard for *portable interceptors* [9]. This standard allows interceptor-based mechanisms to work with any CORBA implementation and application.

3. Java Remote Method Invocation

Given the benefits of RPC interceptors, it is surprising that Java's equivalent to RPC, Java Remote Method Invocation (RMI) [10,11], does not provide built-in support for interceptors. Fortunately, creating custom interceptors for Java RMI is a relatively straightforward process. This section outlines standard Java RMI mechanisms, and the next section shows how we have added custom interceptors to Java RMI.

Figure 2 is a UML class diagram for a set of Java interfaces and classes that represent those required in any RMI-based distributed system. Java interfaces, marked with the «interface» tag, specify operation signatures, but they contain no executable code. The remaining boxes signify Java classes. A line with a closed arrow represents inheritance, while a line with an open arrow indicates reference from one class or interface to another. A line without an arrow represents bidirectional reference between two classes.

Figure 2 shows RMI's built-in interface *java.rmi.Remote* serving as a base interface for any distributed server, represented by application-specific interface *App* in our example. App is a place holder for any application-specific interface that specifies a server's distributed operations.

AppImpl is a server class that implements the operations specified by App. Inheritance establishes the fact that AppImpl satisfies App's contract, i.e., any AppImpl object is an instance of an App object. AppImpl also inherits from (i.e., extends) library class UnicastRemoteObject. The latter class provides networking infrastructure needed to make AppImpl's operations available for distribution.

Once AppImpl has been compiled into a Java class file, RMI's *rmic* compiler utility reads the class file and generates classes AppImpl_Stub, a client-side stub class, and AppImpl_Skel, a server-side skeleton class, as seen in Figure 2. The compiled AppImpl class file gives rmic the information it needs to identify the distributed operations, namely, all operations belonging to parent interfaces from which AppImpl derives, that inherit in turn from interface java.rmi.Remote. In Figure 2 interface App specifies these operations. Rmic generates stub methods for each of these operations into class AppImpl_Stub, and it generates corresponding skeleton code into AppImpl_Skel.

AppClient of Figure 2 is an example of a client class that uses the AppImpl_Stub to make remote method calls. The application logic of AppClient can be coded so that it does not depend on invoking a distributed, AppImpl_Stub method. Instead, AppClient uses some object that implements the App interface. This object may be an actual AppImpl object for a non-distributed system, or it may be an AppImpl_Stub object that also implements the App interface by delegating method calls to a remote AppImpl object.

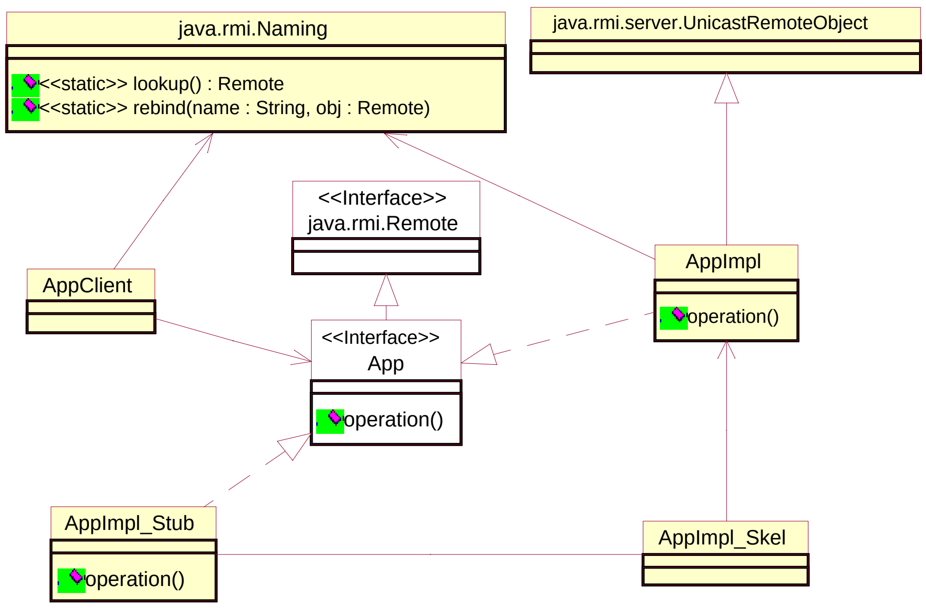


Figure 2: Representative interfaces and classes for a Java RMI-based system

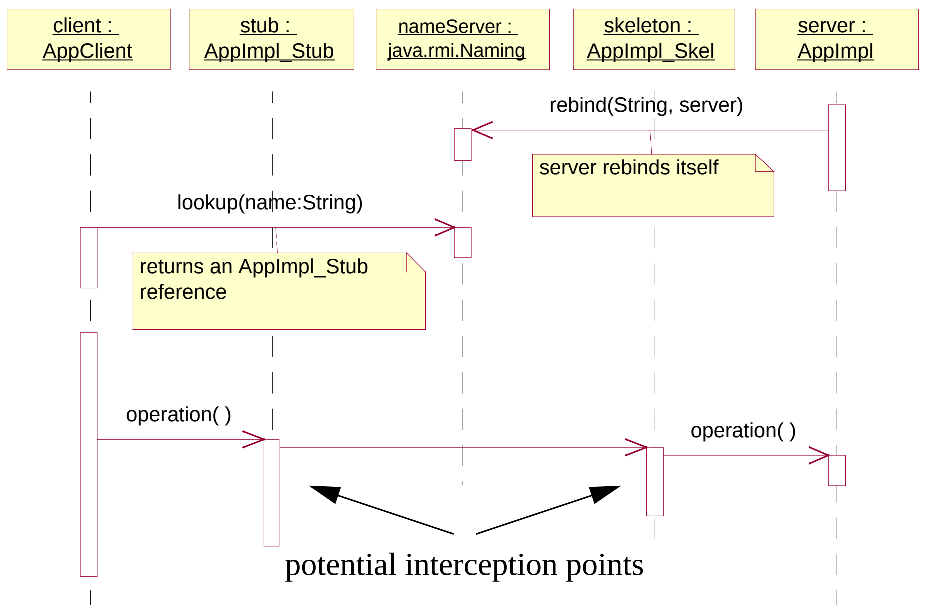


Figure 3: Sequence of steps in connecting an RMI client to a server

The final class of Figure 2, java.rmi.Naming, helps in the initial connection of a client to a server as outlined in the UML sequence diagram of Figure 3. First the server object uses the *Naming.rebind* operation to bind itself to a symbolic name in the Naming service; rebind associates an AppImpl_Skel object with the name. When AppClient invokes *Naming.lookup* to retrieve an object associated with that name that implements the *App* interface, AppClient gets back an AppImpl_Stub object. Thereafter, interactions between AppClient and AppImpl occur largely as they would for any other RPC system. AppClient invokes distributed operations on its AppImpl_Stub object, which the latter forwards via transport to the AppImpl_Skel object for delegation to the AppImpl server.

4. Interceptors for Java RMI

Figure 4 shows a modified version of the class diagram of Figure 2 that illustrates our custom RMI interceptors. Rather than distribute the AppImpl class, we generate and distribute an *AppInterceptor* class that constitutes the server-side interceptor. Our interceptor generator uses Java *reflection* [12] to determine the methods of AppImpl that belong to remote interfaces derived from java.rmi.Remote. Java reflection gives a program such as our generator the ability to inspect a class's ancestor classes and interfaces, along with methods, parameter types and return types. Our generator uses reflection to navigate up from any compiled class derived from java.rmi.Remote to determine its RMI-distributed operations. AppInterceptor's constructor takes an AppImpl object reference as a parameter, and the generator creates an AppInterceptor method for each of AppImpl's distributed methods. An inert version of AppInterceptor simply passes method invocations on to corresponding methods of the AppImpl object, but a custom class derived from AppInterceptor can supply redefinitions of some or all of these server-side interceptor methods.

We now apply *rmic* to AppInterceptor, distributing the server interceptor in addition to the actual server. In fact we can distribute a class derived from AppInterceptor that adds interception-specific methods, such as inter-process synchronization or thread identification operations, to the set of distributed methods. AppImpl and AppClient methods remain unaware of these added interception "back door" methods.

Our generator also generates the client-side interceptor class *AppClientInterceptor* of Figure 4. AppClientInterceptor, like AppInterceptor, has one method per distributed App operation. An inert version of AppClientInterceptor passes method invocations to corresponding methods of

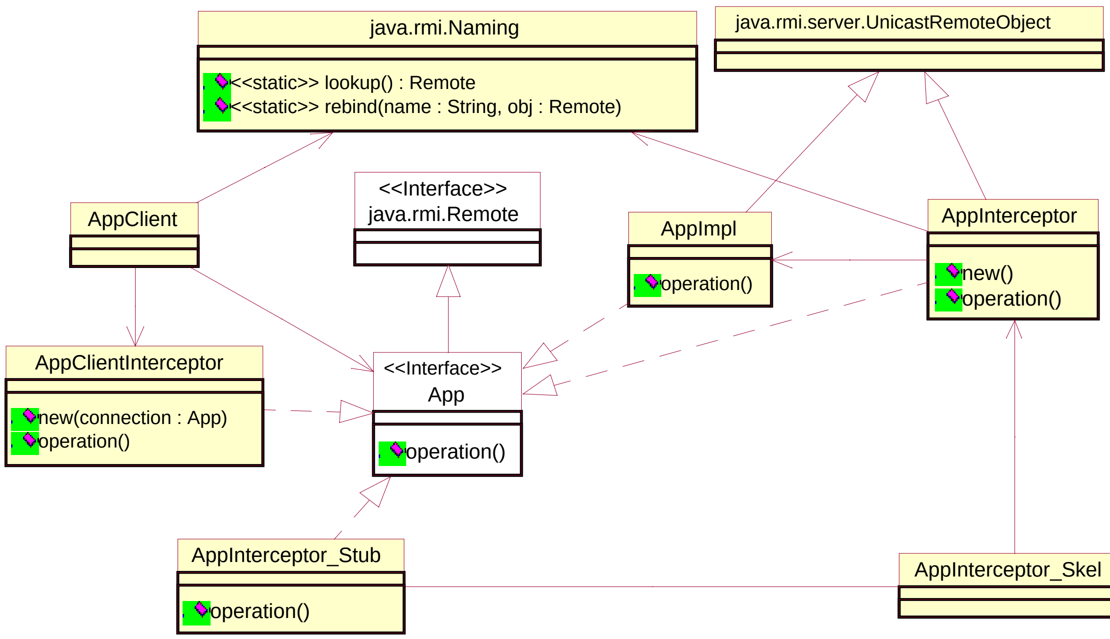


Figure 4: Interfaces and classes for RMI interception

AppInterceptor_Stub. Again we can derive a custom class from AppClientInterceptor, for example to add invocations of the interception "back door" methods of AppInterceptor.

Figure 5 gives the sequence diagram of our approach. The server now constructs an AppInterceptor object that takes its AppImpl object as a constructor parameter; the server passes the AppInterceptor reference to Naming.rebind for distribution. The client now retrieves an AppInterceptor_Stub object reference from Naming.lookup, and the client passes this stub reference to an AppClientInterceptor constructor. Thereafter, the sequence of distributed operation invocations occurs as diagrammed in Figure 5. Invocation flows from AppClient, through AppClientInterceptor, the AppInterceptor_Stub and the transport to AppInterceptor_Skel, on to AppInterceptor and finally to AppImpl. Return results follow the reverse route.

The prototype implementation of our approach requires small source code changes to the original App server and client code. Our prototype explicitly constructs an AppInterceptor object with an AppImpl parameter, and it explicitly constructs an AppClientInterceptor within AppClient code. The latter modification to client code is especially odious, since client-side transparency of interception is very desirable as part of a modular design. A better solution is for a client to invoke an *Abstract Factory Method* [5] to obtain its App interface object. An abstract factory hides construction details from the client, returning some object that implements interface App. By decoupling interceptor and stub construction from AppClient source code, the system regains it modularity and it gains flexibility in selecting from alternative custom interceptors at run time. The result is a very powerful mechanisms for exploring the application space of Java RMI interception.

5. Conclusions and directions

Interceptors have a proven record for dynamic program monitoring and extension. Given their usefulness and their appearance in distributed processing standards such as CORBA, they promise to be equally useful for RMI-based systems. Fortunately, RMI is based on generation of modular distributed communication classes from interface specifications, an approach that is straightforward to extend. Java reflection allows us to automatically generate interceptor shell classes that we can extend through inheritance and dynamic constructor selection.

Our application areas of interest are debugging and profiling. While the Java Platform

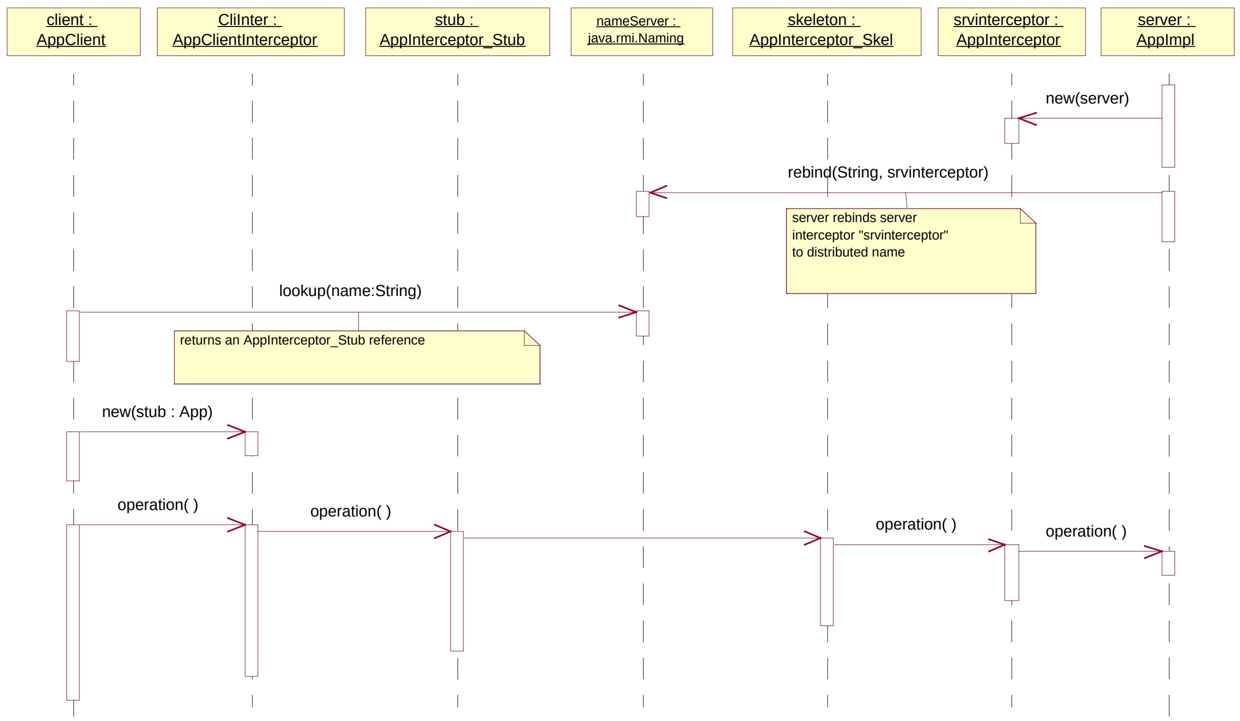

client : AppClient
CliInter : AppClientInterceptor
stub : AppInterceptor_Stub
nameServer : java.rmi.Naming
skeleton : AppInterceptor_Skel
srvinterceptor : AppInterceptor
server : AppImpl
new(server)
rebind(String, srvinterceptor)
server rebinds server interceptor "srvinterceptor" to distributed name
lookup(name:String)
returns an AppInterceptor_Stub reference
new(stub : App)
operation( )
operation( )
operation( )
operation( )

Figure 5: Sequence of steps for RMI interception

Debugger Architecture [13] provides excellent support for debugging individual Java Virtual Machine (JVM) processes, it provides no support for debugging systems distributed over RMI. We are using the RMI interceptor mechanisms described in this paper to explore coordinated debugging of multiple distributed JVMs, with interceptors communicating parameters, results, state information and processing events to a central debugger that is also written in Java.

6. References

1. B. Welch, *Practical Programming in Tcl and Tk*, Third Edition. Upper Saddle River, NJ: Prentice Hall PTR, 1999.
2. T. Curry, “Profiling and Tracing Dynamic Library Usage via Interposing,” Proceedings of the Summer 1994 USENIX Conference. Berkeley, CA: USENIX, June, 1994.
3. J. Rosenberg, *How Debuggers Work*. New York, John Wiley & Sons, 1996.
4. P. Narasimhan, L. Moser and P. M. Melliar-Smith, “Using Interceptors to Enhance CORBA,” IEEE *Computer*, July, 1999, p. 62-68.
5. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
6. *The Common Object Request Broker: Architecture and Specification*, Rev. 2.2, Object Management Group, Framingham, Mass., 1998; ftp://ftp.omg.org/pub/docs/formal/98-07-01.pdf.
7. J. Siegel, *CORBA Fundamentals and Programming*. New York, John Wiley & Sons, 1996.
8. VisiBroker 4 Features and Benefits, http://www.inprise.com/visibroker, 2000.
9. Object Management Group, Portable Interceptors Joint Submission, OMG documents orbos/99-12-02 and orbos/99-12-03, http://www.omg.org.
10. Sun Microsystems, Java Remote Method Invocation, http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.
11. A. Wollrath, J. Waldo and R. Riggs, “Java-Centric Distributed Computing.” IEEE Micro, May/June 1997, p. 44-53.
12. K. Arnold and J. Gosling, *The Java™ Programming Language*, Second Edition. Reading, MA: Addison-Wesley, 1997.
13. Sun Microsystems, Java Platform Debugger Architecture, http://java.sun.com/products/jpda.