CHAPTER II

NRMI

This chapter presents Natural Remote Method Invocation (NRMI): a middleware mechanism that provides a fully-general implementation of call-by-copy-restore semantics for arbitrary linked data structures, used as parameters in remote procedure calls. As a parameter passing semantics, call-by-copy-restore is more natural than traditional call-by-copy, enabling remote calls to behave much like local calls. We discuss in depth the effects of calling semantics for middleware, present scenarios in which NRMI is more convenient to use than regular Java RMI, and describe three efficient implementations of call-by-copy-restore middleware, showing how the lessons of NRMI are reusable in different settings.

2.1 Introduction

Remote Procedure Call (RPC) [10] is one of the most widespread paradigms for distributed middleware. The goal of RPC middleware is to provide an interface for remote services that is as convenient to use as local calls. RPC middleware with *call-by-copy-restore* semantics has been often advocated in the literature, as it offers a good approximation of local execution (*call-by-reference*) semantics, without sacrificing performance. Nevertheless, call-by-copy-restore middleware is not often used to handle arbitrary linked data structures, such as lists, graphs, trees, hash tables, or even non-recursive structures such as a "customer" object with pointers to separate "address" and "company" objects. This is a serious restriction and one that has often been identified. The recent (2002) Tannenbaum and van Steen "Distributed Systems" textbook [83] summarizes the problem and (most) past approaches:

> *... Although [call-by-copy-restore] is not always identical [to call-by-reference], it frequently is good enough. ... [I]t is worth noting that although we can now handle pointers to simple arrays and structures, we still cannot handle the most general case of a pointer to an arbitrary data structure such as a complex graph. Some systems attempt to deal with this case by actually passing the pointer to the server stub and generating special code in the server procedure for using pointers. For example, a request may be sent back to the client to provide the referenced data.*

This chapter addresses exactly the problem outlined in the above passage. We describe an algorithm for implementing call-by-copy-restore middleware that fully supports arbitrary linked structures. The technique is very efficient (comparable to regular *call-by-copy* middleware) and incurs none of the overheads suggested by Tanenbaum and van Steen. A pointer dereference by the server does not generate requests to the client. (This would be dramatically less efficient than our approach, as our measurements show.) Our approach does not "generate special code in the server" for using pointers: the server code can proceed at full speed—not even the overhead of a local read or write barrier is necessary.

We concretized our ideas in the form of Natural Remote Method Invocation (NRMI), with three different implementations. The first implementation is a drop-in replacement for Java RMI; the second enables NRMI in the context of the J2EE platform;

and the third introduces NRMI by employing bytecode engineering to retrofit application classes that use the standard RMI API. In all these implementations, the programmer can select call-by-copy-restore semantics for object types in remote calls as an alternative to the standard call-by-copy semantics of Java RMI. (For primitive Java types the default Java call-by-copy semantics is used.) All the implementations of NRMI call-by-copy-restore are fully general, with respect to linked data structures, but also with respect to arguments that share structure. NRMI is much friendlier to the programmer than standard Java RMI: in most cases, programming with NRMI is identical to non-distributed Java programming. In fact, the call-by-copy-restore implementations in NRMI are guaranteed to offer identical semantics to call-by-reference in the important case of single-threaded clients and stateless servers (i.e., when the server cannot maintain state reachable from the arguments of a call after the end of the call). Since statelessness is a desirable property for distributed systems, anyway, NRMI often offers behavior practically indistinguishable from local calls.

We would be amiss not to mention up front that other middleware services (most notably the DCE RPC standard) have attempted to approximate call-by-copy-restore semantics, with implementation techniques similar to ours. Nevertheless, DCE RPC stops short of full call-by-copy-restore semantics, as we discuss in Section 2.4.2.

In summary, this chapter presents the following insights:

- A clear exposition of different calling semantics, as these pertain to RPC middleware. There is confusion in the literature regarding calling semantics with respect to pointers. This confusion is apparent in the specification and popular implementations of existing middleware (especially DCE RPC, due to its semantic complexity).

- A case for the use of call-by-copy-restore semantics in actual middleware. We argue that such a semantics is convenient to use, easy to implement, and efficient in terms of the amount of transferred data.
- An applied result in the form of three concrete implementations of NRMI. NRMI is a mature and efficient middleware mechanism that Java programmers can adopt on a per case basis as a transparent enhancement of Java RMI. The results of NRMI (call-by-copy-restore even for arbitrary linked structures) can be simulated with RMI (call-by-copy), but this task is complicated, inefficient, and application-specific. In simple benchmark programs, NRMI saves up to 100 lines of code per remote call. More importantly, this code cannot be written without complete understanding of the application's aliasing behavior (i.e., what pointer points where on the heap). NRMI eliminates all such complexity, allowing remote calls to be used almost as conveniently as local calls.

2.2 Background and Motivation

Remote calls in RPC middleware cannot *efficiently* support the same semantics as local calls for data accessed through memory pointers (*references* in Java—we will use the two terms interchangeably). The reason is that efficiently sharing data through pointers (call-by-reference) relies on the existence of a shared address space. The problem is significant because most common data structures in existence (trees, graphs, linked lists, hash tables, and so forth) are heap-based and use pointers to refer to the stored data.

A simple example demonstrates the issues. This will be our main running example throughout the chapter. We will use Java as our demonstration language and Java RMI as the main point of reference in the middleware space. Nevertheless, both Java and Java RMI are highly representative of languages that support pointers and RPC middleware mechanisms, respectively. Consider a simple linked data structure: a binary tree, `t`, storing integer numbers. Every tree node will have three fields, `data`, `left`, and `right`. Consider also

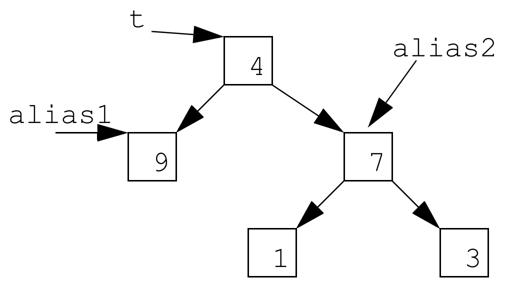


Figure 2-1: A tree data structure and two aliasing references to its internal nodes.

that some of the subtrees are also pointed to by non-tree pointers (aka *aliases*). Figure 2-1 shows an instance of such a tree.

When the tree `t` is passed to a local method that modifies some of its nodes, the modifications affect the data reachable from `t`, `alias1`, and `alias2`. For instance, consider the following method:

```
void foo(Tree tree) {

  tree.left.data = 0;
  tree.right.data = 9;
  tree.right.right.data = 8;
  tree.left = null;
  Tree temp = new Tree(2, tree.right.right, null);
  tree.right.right = null;
  tree.right = temp;

}
```

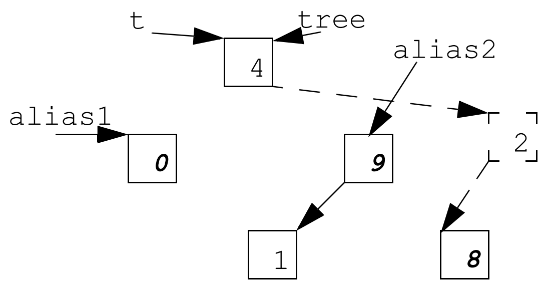


Figure 2-2: A local call can affect all reachable data

(New number values are shown in bold and italic. New nodes and references are dashed. Null references are not shown.)

Figure 2-2 shows the results on the data structure after performing a call `foo(t)` locally. In general, a local call can change all data reachable from a memory reference. Furthermore, all changes will be visible to aliasing references. The reason is that Java has *call-by-value* semantics for all values, including references, resulting into *call-by-reference* semantics for the data pointed to by these references. (From a programming languages standpoint, the Java calling semantics is more accurately called call-by-reference-value. In this chapter, we follow the convention of the Distributed Systems community and talk about "call-by-reference" semantics, although references themselves are passed by value.) The call `foo(t)` proceeds by creating a copy, `tree`, of the reference value `t`. Then every modification of data reachable from `tree` will also modify data reachable from `t`, as `tree` and `t` operate on the same memory space. This behavior is standard in the vast majority of programming languages with pointers.

Consider now what happens when `foo` is a remote method, implemented by a server on a different machine. An obvious solution would be to maintain call-by-reference semantics by introducing "remote references" that can point to data in a different address space, as shown in Figure 2-3.

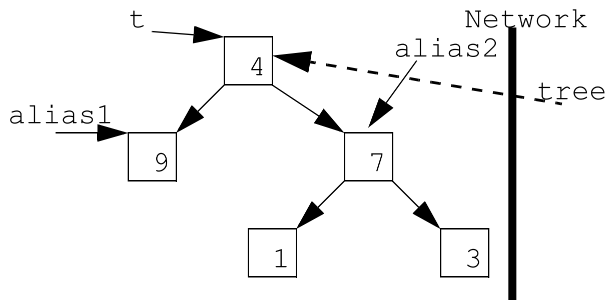


Figure 2-3: Call-by-reference semantics can be maintained with remote references.

Remote references can indeed ensure call-by-reference semantics. Nevertheless, this solution is extremely inefficient. It means that every pointer dereference has to generate network traffic.

Most *object-oriented* middleware (e.g., RMI, CORBA, and so forth and not just traditional RPC) support remote references, which are remotely-accessible objects with unique identifiers; references to them can be passed around similarly to regular local references. For instance, Java RMI allows the use of remote references for subclasses of the `UnicastRemoteObject` class. All instances of the subclass are remotely accessible throughout the network through a Java interface.

Nevertheless, the usual semantics for reference data in RMI calls (and the vast majority of other middleware) is *call-by-copy*. ("Call-by-copy" is really the name used in the Distributed Systems community for *call-by-value*, when the values are complex data structures.) When a reference parameter is passed as an argument to a remote routine, all data reachable from the reference are deep-copied to the server side. The server then operates on the copy. Any changes made to the deep copy of the argument-reachable data are not propagated back to the client, unless the user explicitly arranges to do so (e.g., by passing the data back as part of the return value).

A well-studied alternative of call-by-copy in middleware is *call-by-copy-restore*. Call-by-copy-restore is a parameter passing semantics that is usually defined informally as "having the variable copied to the stack by the caller ... and then copied back after the call, overwriting the caller's original value" [83]. A more strict (yet still informal) definition of call-by-copy-restore is:

Making accessible to the callee a copy of all data reachable by the caller-supplied arguments. After the call, all modifications to the copied data are reproduced on the original data, overwriting the original data values in-place.

Often, existing middleware (notably CORBA implementations through `inout` parameters) support call-by-copy-restore but not for pointer data. Here we discuss what is needed for a fully-general implementation of call-by-copy-restore, per the above definition. Under call-by-copy-restore, the results of executing a remote call to the previously described function `foo` will be those of Figure 2-2. That is, as far as the client is concerned, the call-by-copy-restore semantics is indistinguishable from the call-by-reference one for this example. (As we discuss in Section 2.4, in a single-threaded setting, the two semantics have differences only when the server maintains state that outlives the remote call.)

Supporting the call-by-copy-restore semantics for pointer-based data involves several complications. Our example function `foo` illustrates them:

- call-by-copy-restore has to "overwrite" the original data objects (e.g., `t.right.data` in our example), not just link new objects in the structure reachable from the reference argument of the remote call (`t` in our example). The reason is that, at the client site, the objects may be reachable through other references (`alias2` in our example) and the changes should be visible to them as well.

- some data objects (e.g., node `t.left` before the call) may become unreachable from the reference argument (`t` in our example) because of the remote call. Nevertheless, the new values of such objects should be visible to the client, because at the client site the object may be reachable through other references (`alias1` in our example).
- as a result of the remote call, new data objects may be created (`t.right` after the call in our example), and they may be the only way to reach some of the originally reachable objects (`t.right.left` after the call in our example).

Most of the above complications have to do with aliasing references, i.e., multiple paths for reaching the same heap data. Common reasons to have such aliases include multiple indexing (e.g., the data may be indexed in one way using the tree and in another way using a linked list), caching (storing some recent results for fast retrieval), and others. In general, aliasing is very common in heap-based data, and, thus, supporting it correctly for remote calls is important.

2.3 Supporting Copy-Restore

Having introduced the complications of copy-restore middleware, we now discuss an algorithm that addresses them. The algorithm appears below in pseudo-code and is illustrated on our running example in Figures 2-4 to 2-7.

1. Create a linear map of all objects reachable from the reference parameter. Keep a reference to it.

2. Send a deep copy of the linear map to the server site (this will also copy all the data reachable from the reference argument, as the reference is reachable from the map). Execute the remote procedure on the server.

3. Send a deep copy of the linear map (or a “delta” structure—see Section 2.5) back to the client site. This will copy back all the “interesting” objects, even if they have become unreachable from the reference parameter.

4. Match up the two linear maps so that “new” objects (i.e., objects allocated by the remote routine) can be distinguished from “old” objects (i.e., objects that did exist before the remote call even if their data have changed as a result). Old objects have two versions: original and modified.

5. For each old object, overwrite its original version data with its modified version data. Pointers to modified old objects should be converted to pointers to the corresponding original old objects.

6. For each new object, convert its pointers to modified old objects to pointers to the corresponding original old objects.

The above algorithm reproduces the modifications introduced by the server routine on the client data structures. The interesting part of the algorithm is the automatically keeping track (on the server) of all objects initially reachable by the arguments of a remote method, as well as their mapping back to objects in client memory. The advantage of the algorithm is that it does not impose overhead on the execution of the remote routine. In particular, it completely eliminates the need to trap either the read or the write operations performed by the remote routine by introducing a read or write barrier. Similarly, no data are transmitted over the network during execution of the remote routine. Furthermore, note that supporting call-by-copy-restore only requires transmitting all data reachable from parameters during the remote call (just like call-by-copy) and sending it back after the call ends. This is already quite efficient and will only become more so in the future, when network bandwidth will be much less of a concern than network latency.

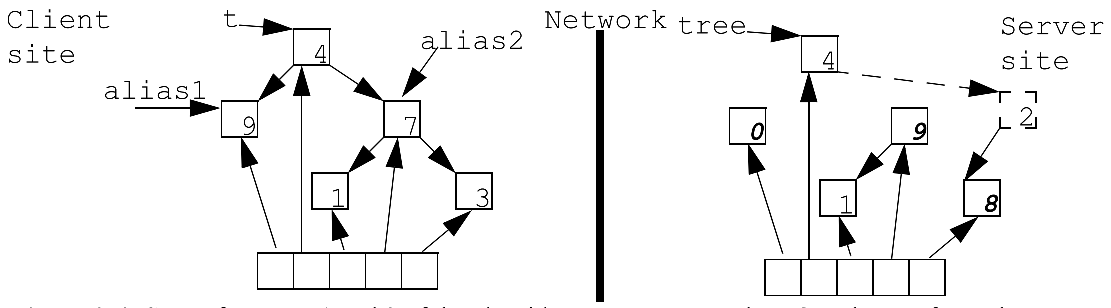


Figure 2-4: State after steps 1 and 2 of the algorithm. Remote procedure `foo` has performed modifications to the server version of the data.

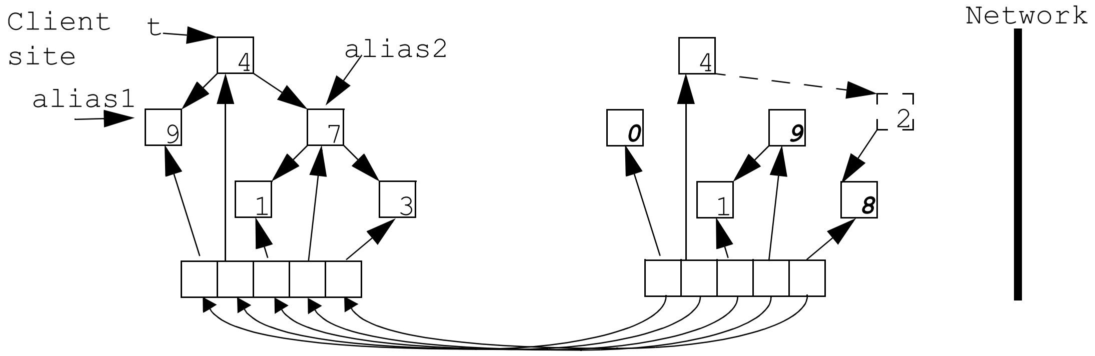


Figure 2-5: State after steps 3 and 4 of the algorithm. The modified objects (even the ones no longer reachable through `tree`) are copied back to the client. The two linear representations are “matched”—i.e., used to create a map from modified to original versions of old objects.

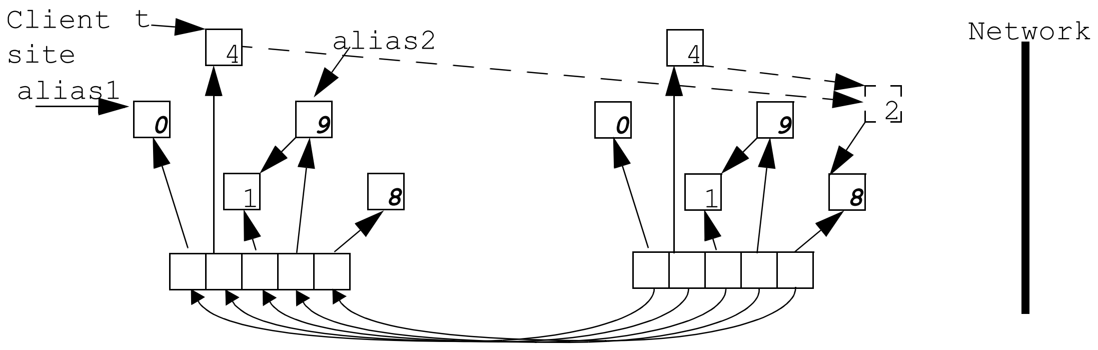


Figure 2-6: State after step 5 of the algorithm. All original versions of old objects are updated to reflect the modified versions.

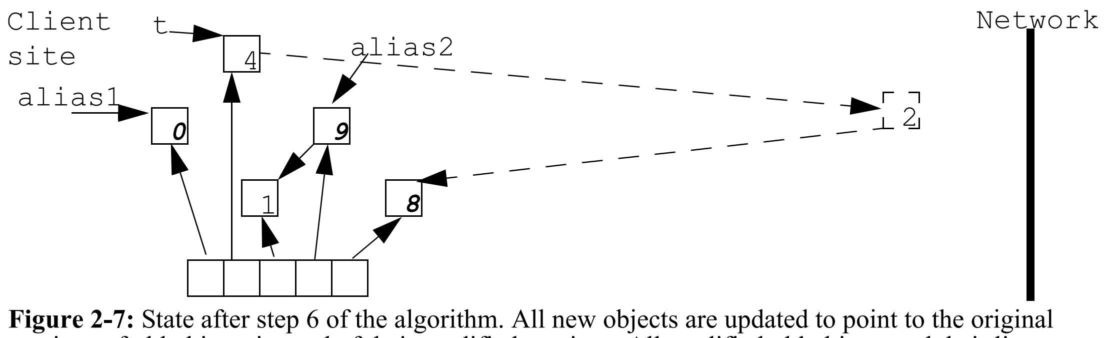


Figure 2-7: State after step 6 of the algorithm. All new objects are updated to point to the original versions of old objects instead of their modified versions. All modified old objects and their linear representation can now be deallocated. The result is identical to Figure 2-3.

2.4 Discussion

2.4.1 Copy-Restore vs. Call-by-Reference

Call-by-copy-restore is a desirable semantics for RPC middleware. Because all mutations performed on the server are restored on the client site, call-by-copy-restore approximates local execution very closely. In fact, one can simply observe that (for a single-threaded client) call-by-copy-restore semantics is identical to call-by-reference if the remote routine is stateless—i.e., keeps no aliases (to the input data) that outlive the remote call. Interestingly, statelessness is a very desirable (for many even indispensable) property for distributed services due to fault tolerance considerations. Thus, a call-by-copy-restore semantics guarantees *network transparency*: a stateless routine can be executed either locally or remotely with indistinguishable results.

The above discussion only considers single-threaded programs. In the case of a multi-threaded client (i.e., caller) network transparency is not preserved. The remote routine acts as a potential mutator of all data reachable by the parameters of the remote call. All updates are performed in an order determined by the middleware implementation. The programmer needs to be aware that the call is remote and that a call-by-copy-restore semantics is used. In the common case, remote calls need to at least execute in mutual exclusion with calls that read/write the same data. If the order of updating matters, call-by-copy-restore can not be used at all: the programmer needs to write code by hand to do the updates in the right order. (Of course, the consideration is for the case of multi-threaded clients—servers can always be multi-threaded and accept requests from multiple client machines without sacrificing network transparency.)

Another issue regarding call-by-copy-restore concerns the use of parameters that share structure. For instance, consider passing the same parameter twice to a remote procedure. Should two distinct copies be created on the remote site or should the sharing of structure be detected and only one copy be created? This issue is not specific to call-by-copy-restore, however. In fact, regular call-by-copy middleware has to answer the same question. Creating multiple copies can be avoided using exactly the same techniques as in call-by-copy middleware (e.g., Java RMI)—the middleware implementation can notice the sharing of structure and replicate the sharing in the copy. Unfortunately, there has been confusion on the issue. Based on existing implementations of call-by-copy-restore for primitive (non-pointer) types, an often repeated mistaken assertion is that call-by-copy-restore semantics implies that shared structure results in multiple copies [82][83][93].

2.4.2 DCE RPC

The DCE RPC specification [63] is the foremost example of a middleware design that tries to enable distributed programming in a way that is as natural as local programming. The most widespread DCE RPC implementation nowadays is that of Microsoft RPC, forming the base of middleware for the Microsoft operating systems. Readers familiar with DCE RPC may have already wondered if the specification for pointer passing in DCE RPC is not identical to call-by-copy-restore. The DCE RPC specification stops one step short of call-by-copy-restore semantics, however.

DCE RPC supports three different kinds of pointers, only one of which (*full pointers*) supports aliasing. DCE RPC full pointers, declared with the `ptr` attribute, can be safely aliased and changed by the callee of a remote call. The changes will be visible to the caller, even through aliases to existing structure. Nevertheless, DCE RPC only guarantees correct updates of aliased data for aliases that are declared in the parameter lists of a remote call.[1] In other words, for pointers that are not reachable from the parameters of a remote call, there is no guarantee of correct update.

In practical terms, the lack of full alias support in the DCE RPC specification means that DCE RPC implementations do not support call-by-copy-restore semantics for linked data structures. In Microsoft RPC, for instance, the calling semantics differs from call-by-copy-restore when data become unreachable from parameters after the execution of a remote call. Consider again our example from Section 2.2. Figure 2-2 is reproduced here as Figure 2-8 for easy reference.

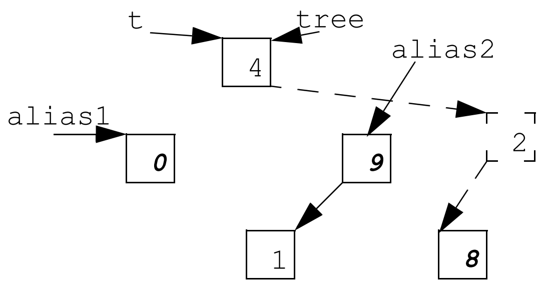


Figure 2-8: Changes after execution of method.

The remote call that operates on argument `t`, changes the data so that the former objects `t.left` and `t.right` are no longer reachable from `t`. Under call-by-copy-restore semantics, the changes to these objects should still be restored on the caller site (and thus made visible to `alias1` and `alias2`). This does not occur under DCE RPC, however. The effects of statements

1. The specification reads "*For both out and in, out parameters, when full pointers are aliases, according to the rules specified in Aliasing in Parameter Lists* [these rules read*: If two pointer parameters in a parameter list point at the same data item*]*, the stubs maintain the pointed-to objects such that any changes made by the server are reflected to the client for all aliases.*"

```
tree.left.data = 0;
tree.right.data = 9;
tree.right.right = null;
```

would be disregarded on the caller site. Figure 2-9 shows the actual results for DCE RPC.

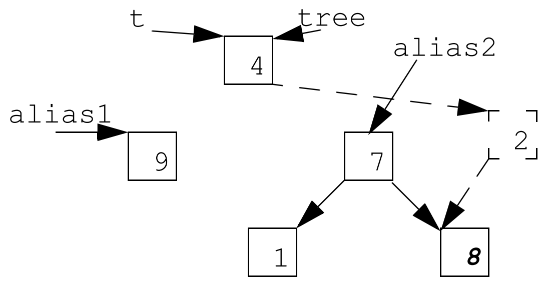


Figure 2-9: Under DCE RPC, the changes to data that became unreachable from `t` will not be restored on the client site.

2.5 NRMI Implementations

We now describe the particulars of implementing NRMI. Despite the fact that our implementations are Java specific, the insights are largely language independent. Our call-by-copy-restore algorithm can be applied to any other distribution middleware that supports pointers.

NRMI currently has three implementations, each applicable to different programming environments and scenarios. The implementation in the form of a full, drop-in replacement for Java RMI demonstrates how this standard middleware mechanism for the Java language can be transparently enhanced with call-by-copy-restore capacities. However, introducing a new feature into the implementation of a standard library of a mainstream programming language is a significant undertaking, requiring multiple stakeholders in the Java technology to reach a consensus. Therefore, our other two implementations provide Java programmers with call-by-copy-restore capacities without having to change any of the standard Java libraries. One implementation takes advantage of the extensible application server architecture offered by JBoss [68] to introduce NRMI as a pair of client/server interceptors. Another introduces NRMI by retrofitting the bytecodes of application classes that use the standard RMI API. Having to work around the inability to change the RMI runtime libraries, these latter two solutions are not always as efficient as the drop-in replacement one but offer interesting insights on how new middleware features can be introduced transparently. Therefore, we limit our discussion on various optimization issues of NRMI to the RMI drop-in replacement implementation only.

2.5.1 A Drop-in Replacement of Java RMI

2.5.1.1 Programming Interface

Our drop-in replacement for Java RMI supports a strict superset of the RMI functionality by providing call-by-copy-restore as an additional parameter passing semantics to the programmer. This implementation follows the design principles of RMI in having the programmer decide the calling semantics for object parameters on a per-type basis. In brief, indistinguishably from RMI, NRMI passes instances of subclasses of `java.rmi.server.UnicastRemoteObject` by-reference and instances of types that implement `java.io.Serializable` by-copy. Values of primitive types are passed by-copy ("by-value" in programming languages terminology). That is, just like in regular RMI, the following definition makes instances of class `A` be passed by-copy to remote methods.

```
//Instances will be passed by-copy by NRMI
class A implements java.io.Serializable {...}
```

Our drop-in implementation introduces a marker interface `java.rmi.Restor-able`, which allows the programmer to choose the by-copy-restore semantics: parameters whose class implements `java.rmi.Restorable` are passed by copy-restore. For example:

```
//Instances passed by-copy-restore by NRMI
class A implements java.rmi.Restorable {...}
```

`java.rmi.Restorable` extends `java.io.Serializable`, reflecting the fact that call-by-copy-restore is basically an extension of call-by-copy. In particular, “restorable” classes have to adhere to the same set of requirements as if they were to be passed by-copy—i.e., they have to be serializeable by Java Serialization [80].

In the case of JDK classes, `java.rmi.Restorable` can be implemented by their direct subclasses as follows:

```
//Instances passed by-copy-restore by NRMI
class RestorableHashMap extends java.util.HashMap
                    implements java.rmi.Restorable {...}
```

In those cases when subclassing is not possible, a delegation-based approach can be used as follows:

```
//Instances passed by-copy-restore by NRMI
class SetDelegator implements java.rmi.Restorable {
  java.util.Set _delegatee;
  //expose the necessary functionality
  void add (Object o) { _delegatee.add (o); }
  ...
}
```

Declaring a class to implement `java.rmi.Restorable` is all that is required from the programmer: NRMI will pass all instances of such classes by-copy-restore whenever they are used in remote method calls. The NRMI runtime handles the restore phase of the algorithm totally transparently to the programmer. This saves lines of tedious and error-prone code as we illustrate in Section 5.2.

In order to make NRMI easily applicable to existing types (e.g., arrays) that cannot be changed to implement `java.rmi.Restorable`, we adopted the policy that a reachable, serializable sub-object is passed by-copy-restore, if its parent object implements `java.rmi.Restorable`. Thus, if a parameter is of a "restorable" type, everything reachable from it will be passed by-copy-restore (assuming it is serializable, i.e., it would otherwise be passed by copy).

It is worth noting that Java fits the bill as a language for demonstrating the benefits of call-by-copy-restore middleware because of its local method call semantics. In local Java method calls, all primitive parameters are passed by-copy ("by-value" using programming languages terminology). This is identical behavior with remote calls in Java using either standard RMI or NRMI. With NRMI we also add call-by-copy-restore semantics for reference types, thus making the behavior of remote calls be (almost) identical to local calls even for non-primitive types. Thus, with NRMI, distributed Java programming is remarkably similar to local Java programming.

2.5.1.2 Implementation Insights

Having introduced the programming interface offered by our drop-in replacement implementation of NRMI, we now describe it in greater detail. We analyze one-by-one each of the major steps of the algorithm presented in Section 2.3.

Creating a linear map

Creating a linear map of all objects reachable from the reference parameter is obtained by tapping into the Java Serialization mechanism. The advantage of this approach is that we get a linear map almost for free. The parameters passed by-copy-restore have to be serialized anyway, and the process involves an exhaustive traversal of all the objects reachable from these parameters. The linear map that we need is just a data structure storing references to all such objects in the serialization traversal order. We get this data structure with a tiny change to the serialization code. The overhead is minuscule and only present for call-by-copy-restore parameters.

Performing remote calls

On the remote site, a remote method invocation proceeds exactly as in regular RMI. After the method completes, we marshall back linear map representations of all those parameters whose types implement `java.rmi.Restorable` along with the return value if the method has one.

Updating original objects

Correctly updating original reference parameters on the client site includes matching up the new and old linear maps and performing a traversal of the new linear map. Both step 5 and step 6 of the algorithm are performed in a single depth-first traversal by just performing the right update actions when an object is first visited and last visited (i.e., after all its descendants have been traversed).

Optimizations

The following two optimizations can be applied to an implementation of NRMI in order to trade processing time for reduced bandwidth consumption. First, instead of sending the linear map over the network, we can reconstruct it during the un-serialization phase on the server site of the remote call. Second, instead of returning the new values for all objects to the caller site, we can send just a “delta” structure, encoding the difference between the original data and the data after the execution of the remote routine. In this way, the cost of passing an object by-copy-restore and not making any changes to it is almost identical to the cost of passing it by-copy. Our implementation applies the first optimization, while the second will be part of future work.

2.5.2 NRMI in the J2EE Application Server Environment

A J2EE [78] application server is a complex standards-conforming middleware platform for development and deployment of component-based Java enterprise applications. These applications consist of business components called Enterprise JavaBeans (EJBs). Application servers provide an execution environment and standard means of accessing EJBs by both local and remote clients. To accomplish that, an EJB can support a local interface for clients, collocated with it in the same JVM, and a remote interface for clients accessing it from different address spaces. With some designs, it can be desirable to be able to treat local and remote accesses uniformly, and call-by-copy-restore can bridge the differences between the local and remote parameters passing semantics. For example, the developer could select call-by-copy-restore semantics on a per method basis if it makes sense to do so from the design perspective. The NRMI semantics is also a great asset in the task of automatic transformation of regular Java classes into EJBs, as, for example, in our GOTECH framework described in Chapter III.

We have implemented NRMI in the application server environment of JBoss taking advantage of its extensible architecture [68]. JBoss is an extensible, open-ended, and dynamically-reconfigurable application server that follows the open source development model. JBoss employs the Interceptor pattern, which enables transparent addition and automatic triggering of services [72] and has become a common extensibility-enhancing mechanism in complex software systems. Indeed, the Interceptor pattern enables the programmer to extend the functionary of such systems without having to understand their inner workings. Informally, a JBoss interceptor is a piece of functionality that gets inserted into the client-server communication path, which gets intercepted in both directions: the client's requests to the server and the server's replies. JBoss interceptors intercept a remote call with the purpose of examining and, in some cases, modifying its parameters or return value and come in two varieties: client and server, specifying their actual deployment and execution locations. JBoss provides flexible mechanisms for creating and deploying interceptors and broadly employs them to implement a large subset of its core functionality such as security and transactions.

Our support for NRMI in JBoss consists of a programming interface, enabling the programmer to choose call-by-copy-restore semantics on a per method basis, and an implementation, consisting of a pair of client server interceptors. Because this implementation works on top of regular RMI, it cannot introduce a new Java marker interface for copy-restore parameters and must follow a different approach. We introduced a new XDoclet [103] annotation `method-parameters copy-restore`, specifying that all reference parameters of a remote method are to be passed by copy-restore. The following code example shows how the programmer can use this new annotation.

```
/**
 * @ejb:interface-method view-type="remote"
 * @jboss:method-parameters copy-restore="true"
 */
public void foo (Reference1 ref1, int i, Reference2 ref2) { ... }
//both ref1 and ref2 will be passed by-copy-restore
```

Note that, in this implementation, it is impossible to enable the programmer to specify call-by-copy-restore semantics for individual parameters: the copy-restore is a per-method annotation and applies to all reference parameters of a remote method.

From the implementation perspective, the NRMI interceptors are subclasses of `org.jboss.proxy.Interceptor` and `org.jboss.ejb.plugins.AbstractInterceptor` classes for the client and the server portions of the code, respectively. The NRMI interceptors are invoked only for those methods specified as having the call-by-copy-restore semantics. It took only about 100 lines of Java code to supply the logic of both NRMI interceptors. This number excludes the actual NRMI algorithm implementation (another 700 lines of code). Below is the simplified code for the `invoke` methods of the NRMI client and server interceptors.

```
//in NRMI client interceptor
public InvocationResponse invoke(Invocation invocation)
                                       throws Throwable {

Object[] arguments = invocation.getArguments();

//create linear map representations
//for copy-restore arguments
Object[][] linearRepresentations =
            NRMI.createLinearRepresentations(arguments);
...
//pass the invocation to the next interceptor in the chain
InvocationResponse response =
                            getNext().invoke(invocation);

Object[][] newLinearRepresentations =
           (Object[][])response.getAttachment(LINEAR_MAP);

//after the invocation, perform the restore
//for copy-restore args
NRMI.performRestore(newLinearRepresentations,
                    linearRepresentations);

return response;

}



//in NRMI server interceptor
public InvocationResponse invoke(Invocation invocation)
                                       throws Exception {

Object[][] linearReps =
        NRMI.createLinearRep(invocation.getArguments());

InvocationResponse response =
                            getNext().invoke(invocation);

response.addAttachment(LINEAR_MAP, linearReps);

return response;

}
```

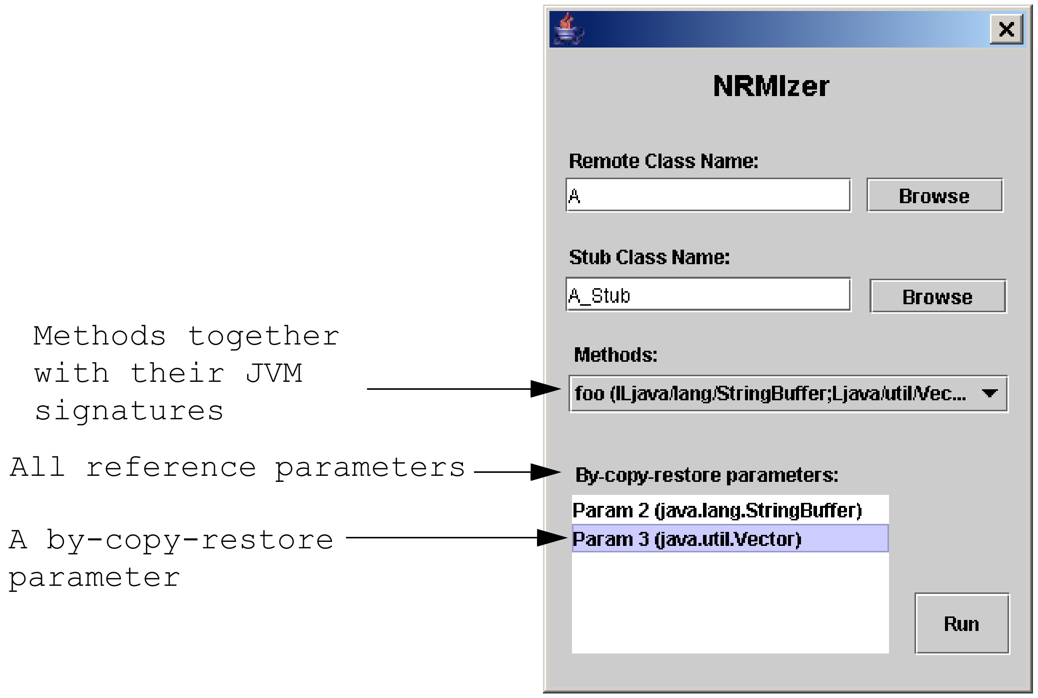
2.5.3 Introducing NRMI through Bytecode Engineering

In some development environments, the programmer could find beneficial the ability to use the call-by-copy-restore semantics on top of a standard unmodified middleware implementation, supporting only the standard call-by-copy semantics. Furthermore, that environment might not provide any built-in facilities for flexible functionality enhancement such as interceptors. For example, the J-Orchestra automatic partitioning system, described in Chapter IV, has as one of its primary design objectives the ability to execute partitioned programs using a standard RMI middleware implementation. By default, J-Orchestra uses the RMI call-by-reference semantics (remote reference) to emulate a shared address space for the partitioned programs. However, as Figure 2-3 shows, any access to a remote object through a remote reference incurs network overhead. Therefore, a program partitioned with J-Orchestra can derive substantial performance benefits by using the call-by-copy-restore semantics in some of its remote calls. It is exactly for these kind of scenarios that we developed our approach for introducing NRMI by retrofitting the bytecodes of application classes that use the standard RMI API.

Prior research has employed bytecode engineering for modifying the default Java RMI semantics with the goal of correctly maintaining thread identity over the network [90][98]. In our implementation, we follow a similar approach that transparently enables the call-by-copy-restore semantics for remote calls that use regular Java RMI.

2.5.3.1 User View: NRMIzer

Our GUI-enabled tool is called NRMIzer. Figure 2-10 shows the tool's GUI. As input, the tool takes two application classes that use the Java RMI API: a remote class (i.e., implementing a remote interface) and its RMI stub. An RMI stub is a client site class that serves as a proxy for its corresponding remote class (i.e., located on a remote server). Under Sun's JDK, stubs are generated in binary form by running the rmic tool against a remote class. The reason why the user has to specify the names of both a remote class and its RMI stub is the possibility of polymorphism in the presence of incomplete program knowledge. Since a stub might be used to invoke methods on a subclass of the remote class from which it was generated, the appropriate transformations must be made to all possible invocations of the remote method through any of the stubs. NRMIzer shows a list of all methods imple-


Methods together with their JVM signatures
All reference parameters
A by-copy-restore parameter

NRMIzer

Remote Class Name: A

Stub Class Name: A_Stub

Methods: foo (ILjava/lang/StringBuffer;Ljava/util/Vec...

By-copy-restore parameters:
Param 2 (java.lang.StringBuffer)
Param 3 (java.util.Vector)

Figure 2-10: NRMIzer GUI.

mented by a selected class, displayed together with their JVM signatures. For each method, the tool also shows a list of its reference parameters. The programmer then selects these parameters individually, conveying to the tool that they are to be passed by-copy-restore.

2.5.3.2 Implementation Specifics: Backend Engine

The backend engine of NRMIzer retrofits the bytecode of a remote class and its RMI stub to enable any reference parameter of a remote method to be passed by-copy-restore. To accomplish the by-copy-restore semantics on top of regular RMI, the tool adds extra code to both the remote class and its stub for each remote method that has any by-copy-restore parameters. Consider the following remote method `foo` taking as parameter an `int` and a `Ref` and returning a `float`. We want to pass its `Ref` parameter by copy-restore.

```
//original remote method foo
//want to pass Ref by copy-restore
public float foo(int i, Ref r) throws RemoteException{...}
```

We show the transformations performed on the stub code, running on the client, next.

```
//change the body of foo as follows (slightly simplified)
public float foo (int i, Ref r) throws RemoteException {
   Object[] linearMap = NRMI.computeLinearMap (r);
   //invoke foo__nrmi remotely
   //NRMIReturn encapsulates both
   //linear maps and the return value of foo
   NRMIReturn ret = foo__nrmi (i, r);
   Object[]newLinearMap = ret.getLinearMap();
   NRMI.performRestore(linearMap, newLinearMap);
   //extract the original return value
   return ((Float)ret.getReturnValue()).floatValue();
}
```

On the server side, the method `foo__nrmi` computes a linear map for the `Ref` parameter, invokes the original method `foo`, and packs both the return `float` value of `foo` and the linear map into a holder object of type `NRMIReturn`. The class `NRMIReturn` encapsulates the original return value of a remote method along with the linear representations of copy-restore parameters. All special-purpose NRMI methods that NRMIzer adds to the remote and stub classes use `NRMIReturn` as their return type.

As far as the runtime deployment is concerned, several classes, implementing the NRMI algorithm, have to be added to the original RMI program. These NRMI runtime classes can either be deployed as a separate jar file or bundled together with the original program's classes.

2.6 Conclusion

In this chapter, we presented NRMI, a middleware mechanism that provides a fully-general implementation of call-by-copy-restore semantics for arbitrary linked data structures, used as parameters in remote procedure calls. We discussed the effects of calling semantics for middleware, explained how our algorithm works, and described three different implementations of call-by-copy-restore middleware. In Chapter V we further discuss various applicability issues of NRMI, present several examples of Java programs in which NRMI is more convenient to use than regular Java RMI, and present detailed performance measurements of our drop-in RMI replacement implementation, proving that NRMI can be implemented efficiently enough for real world use.