NRMI: Natural and Efficient Middleware

Eli Tilevich and Yannis Smaragdakis
Center for Experimental Research in Computer Systems (CERCS), College of Computing, Georgia Tech
{tilevich, yannis}@cc.gatech.edu

Abstract

We present NRMI: a drop-in replacement for Java RMI that supports call-by-copy-restore semantics for arbitrary linked data structures, in addition to regular call-by-copy semantics. Call-by-copy-restore middleware is more natural to use than traditional call-by-copy RPC mechanisms, enabling distributed calls to behave much like local calls. We discuss in depth the effect of calling semantics for middleware, describe how call-by-copy-restore middleware can be implemented efficiently, and show examples of Java programs where NRMI is more convenient than regular Java RMI.

1. Introduction

Remote Procedure Call (RPC) 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, like lists, graphs, trees, hash tables, or even non-recursive structures like a "customer" object with pointers to separate "address" and "company" objects. This is a serious restriction and one that has been often identified. The recent (2002) Tanenbaum and van Steen "Distributed Systems" textbook [17] 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 paper addresses exactly the problem outlined in the above passage. We describe an algorithm for implementing call-by-copy-restore middleware so that arbitrary linked structures are fully supported. The technique is very efficient (comparable to regular *call-by-copy* middleware)—none of the overheads suggested by Tanenbaum and van Steen are incurred. 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.) We do 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 implemented our ideas in the form of *NRMI (Natural Remote Method Invocation)*. NRMI is a modified version of the Java RMI, such that the user can select call-by-copy-restore semantics for object types in remote calls, in addition to the standard call-by-copy semantics of Java RMI. (For primitive Java types the default Java call-by-copy semantics is used.) The implementation of call-by-copy-restore in NRMI is fully general, with respect to linked data structures, but also with respect to arguments that share structure. NRMI is much friendlier to the user than standard Java RMI: in most cases, programming with NRMI is identical to non-distributed Java programming. In fact, the call-by-copy-restore implementation in NRMI is guaranteed to offer identical semantics to call-by-reference in the important case when single-threaded clients and stateless servers are used (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 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 4.2.

This paper makes the following contributions:

- 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 NRMI. NRMI is a mature and efficient middleware implementation 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. 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, etc.) 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 paper. 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 typical 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 that some of the subtrees are also pointed to by non-tree pointers (aka *aliases*). An instance of such a tree is shown in Figure 1.

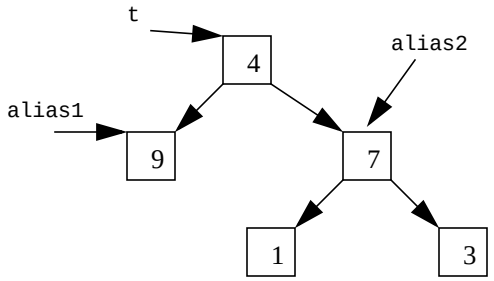


Figure 1: A tree data structure and two aliasing references to its internal nodes.

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

```
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;
}
```

If a call `foo(t)` is performed locally, the results on the data structure will be those shown in Figure 2.

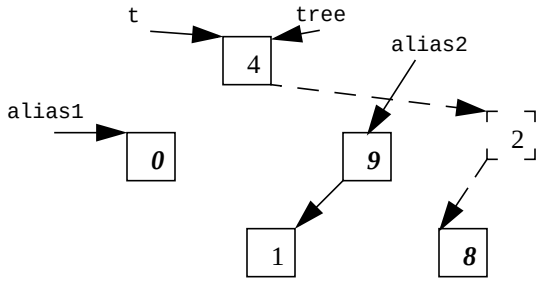


Figure 2: A local call can affect all reachable data

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

In other words, a local call can change all data reachable from a memory reference. 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 paper, 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 that allow pointers.

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

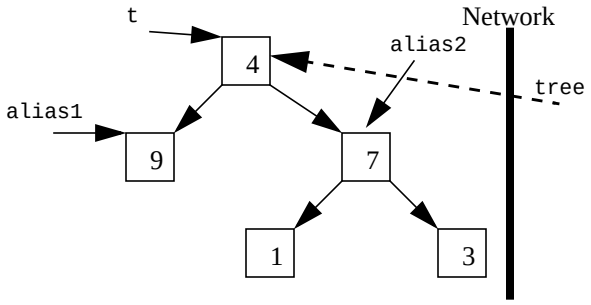


Figure 3: Call-by-reference semantics can be maintained with remote references.

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, etc. and not just traditional RPC) allow the use of remote references, as remotely-accessible objects have unique identifiers and references to them can be passed around like 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" [17]. 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. That is, as far as the client is concerned, the call-by-copy-restore semantics is indistinguishable from a call-by-reference semantics for this example. (As we discuss in Section 4, the two semantics have differences only when the server maintains state that outlives the remote call.)

There are several complications in supporting the call-by-copy-restore semantics for pointer-based data. Our example function `foo` illustrates them:

- call-by-copy-restore has to "overwrite" the original data (e.g., `t.right.data` in our example), not just link new data 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 data may be reachable through other references (`alias2` in our example) and the changes should be visible to them as well.
- some data (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 data should be visible to the client, because at the client site the data may be reachable through other references (`alias1` in our example).
- as a result of the remote call, new data 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 data (`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), etc. Aliasing is very common in heap-based data and supporting it correctly for remote calls is important.

3. Supporting Copy-Restore

Having seen 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 4-7.

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 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, there is no 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.

4. Discussion

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 probably not be used at all. (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. 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 into multiple copies [16][17][21].

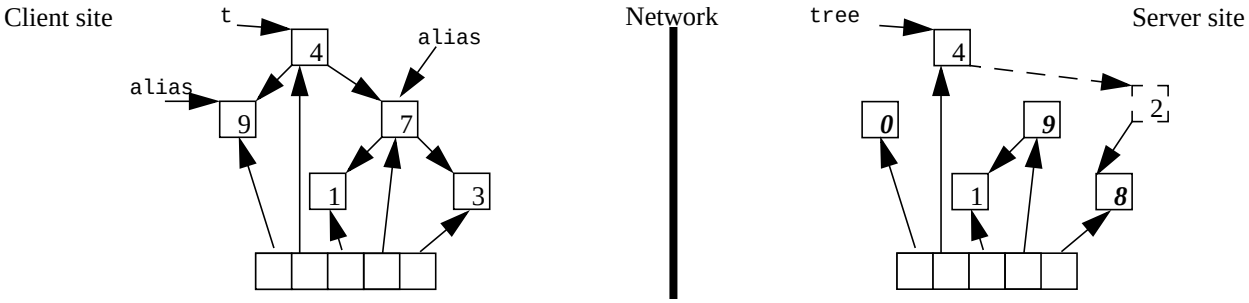


Figure 4: State after steps 1 and 2 of the algorithm. Remote procedure `foo` has performed modifications to the server version of the data.

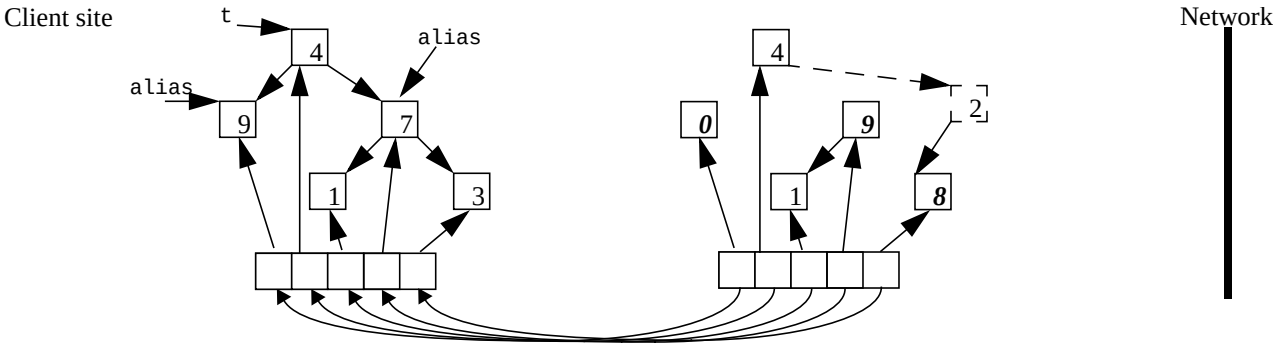


Figure 5: State after steps 3 and 4 of the algorithm. The modified objects (even the ones no longer reachable through `tree`) are copied back to the client. The two linear representations are "matched"—i.e., used to create a map from modified to original versions of old objects.

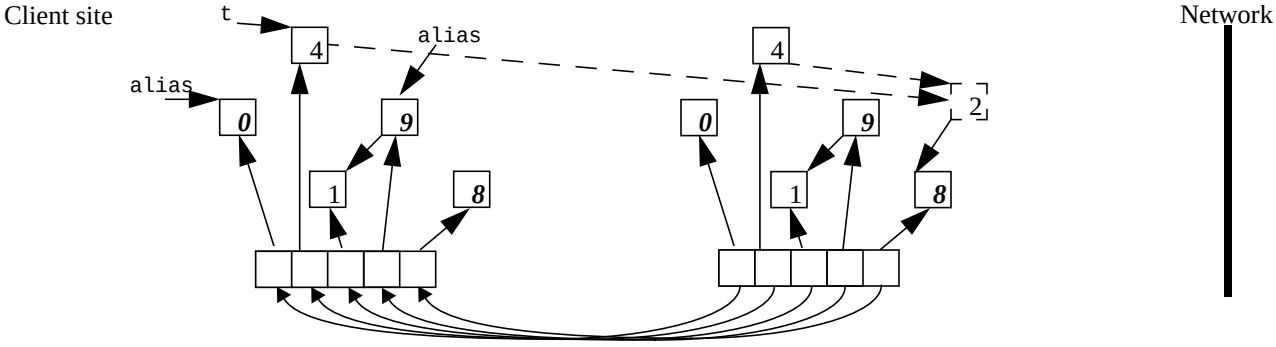


Figure 6: State after step 5 of the algorithm. All original versions of old objects are updated to reflect the modified versions.

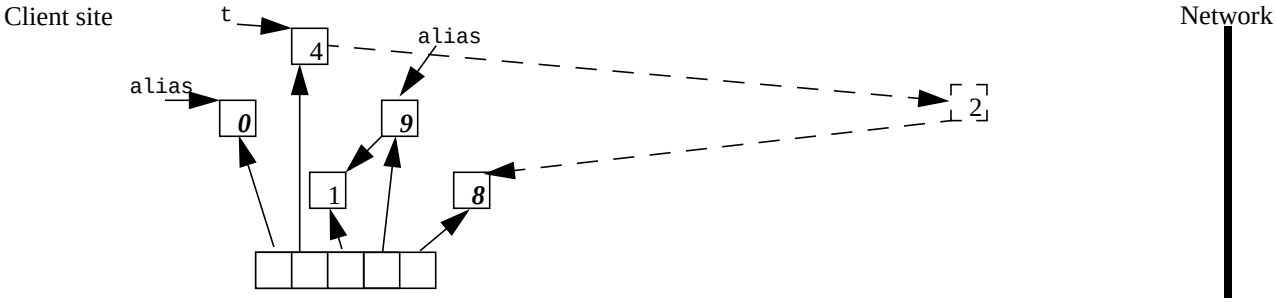


Figure 7: State after step 6 of the algorithm. All new objects are updated to point to the original versions of old objects instead of their modified versions. All modified old objects and their linear representation can now be deallocated. The result is identical to Figure 3.

4.2. DCE RPC

The DCE RPC specification [10] 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. Figure 2 is reproduced here as Figure 8 for easy reference.

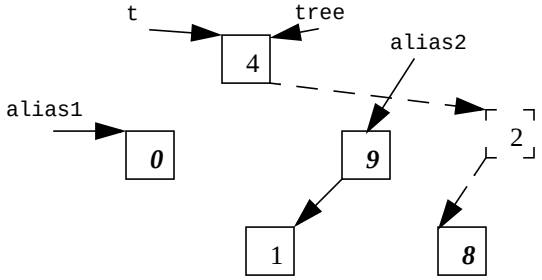


Figure 8: Changes after execution of method.

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

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

would be disregarded on the caller site. The actual results for DCE RPC are shown in Figure 9.

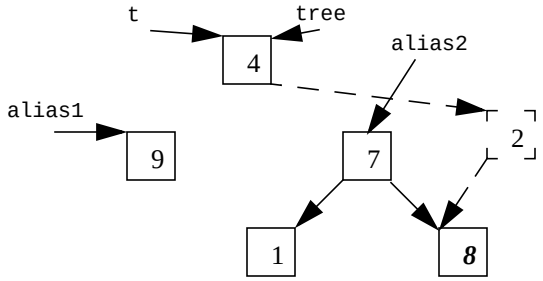


Figure 9: Under DCE RPC, the changes to data that became unreachable from `t` will not be restored on the client site.

4.3. Usability: Copy-Restore vs Call-by-Copy

Compared to call-by-copy, the main benefits of call-by-copy-restore semantics are in usability. Call-by-copy-restore simulates very closely the local execution semantics, as discussed in Section 4.1. Clearly, the behavior of call-by-copy-restore can be achieved by using call-by-copy and adding application-specific code to register and re-perform any updates necessary. Nevertheless, this has several disadvantages:

- The programmer has to be aware of all aliases, in order be able to update the values changed during the remote call, even if the changes are to data that became unreachable from the original parameters.
- The programmer needs to write extra code to perform the update. This code can be long for complex updates (e.g., up to 100 lines per remote call for the microbenchmarks we discuss in Section 5.3).
- The programmer cannot perform the updates without full knowledge of what the server code changed. That is, the changes to the data have to be part of the protocol between the server programmer and the client programmer. This complicates the remote interfaces and specifications.

As we discuss in Section 2, a call-by-copy-restore semantics is most valuable in the presence of aliased data. There are several common implementation techniques in mainstream programming languages that result into alias-

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.*"

ing. All of these techniques produce code that is more convenient to write using call-by-copy-restore middleware than by call-by-copy middleware. Specific examples include:

- *Common GUI patterns like model-view-controller.* Most GUI toolkits register multiple views, all of which correspond to a single model object. That is, all views alias the same model object. An update to the model should result in an update to all of the views. Such an update could be the result of a remote call.

  A variant of this pattern occurs when GUI elements (menus, toolbars, etc.) hold aliases to program data that can be modified. The reason for multiple aliasing is that the same data may be visible in multiple toolbars, menus, etc. or that the data may need to be modified programmatically with the changes reflected in the menu or toolbar. For example, we distribute with NRMI a modified version of one of the Swing API example applications. We changed the application to be able to display its text strings in multiple languages. The change of language is performed by calling a remote translation server when the user chooses a different language from a drop-down box. (That is, the remote call is made in the event dispatching thread, conforming to Swing thread programming conventions.) The remote server accepts a vector of words (strings) used throughout the graphical interface of the application and translates them between English, German and French. The updated list is restored on the client site transparently and the GUI is updated to show the translated words in its menus, labels, etc. The distributed version code only has two tiny changes compared to local code: a single class needs to implement `java.rmi.Restorable` and a method has to be looked up using a remote lookup mechanism before getting called. In contrast, the version of the application that uses regular Java RMI, has to use a more complex remote interface for getting back the changed data and the programmer has to write code in order to perform the update.

- *Multiple indexing*. Most applications in imperative programming languages create some multiple indexing scheme for their data. For example, a business application may keep a list of the most recent transactions performed. Each transaction, however, is likely to also be retrievable through a reference stored in a customer's record, through a reference from the daily tax record object, etc. Similarly, every customer may be retrievable from a data structure ordered by zip code, and from a second data structure ordered by name. All of these references are aliases to the same data (customers, business transactions). NRMI allows such references to be updated correctly as a result of a remote call (e.g., an update of purchase records from a different location, or a retrieval of a customer's address from a central database), in much the same way as they would be updated if the call were local.

5. NRMI

We now describe the particulars of NRMI design and implementation. Despite the fact that our implementation is 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.

5.1. NRMI Programming Interface

NRMI is a drop-in replacement for Java RMI that supports a strict superset of the RMI functionality by providing call-by-copy-restore as an additional parameter passing semantics to the programmer. NRMI 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 subclasses of `java.rmi.server.UnicastRemoteObject` by-reference and types that implement `java.io.Serializable` by-copy. 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 {...}
```

NRMI introduces a marker interface `java.rmi.Restorable` to allow 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 [14].

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

```
//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.

```
//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 restore phase of the algorithm is hidden from the programmer, being handled completely by the NRMI runtime. This saves lines of tedious and error-prone code as we illustrate in Section 4.3.

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 call-by-copy).

It is worth noting that Java is a good language for demonstrating the benefits of call-by-copy-restore middleware because of the local Java 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 Java 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.

5.2. Implementation Insights (High-Level)

Having introduced the programming interface offered by NRMI, we now describe our implementation in greater detail. We analyze one-by-one each of the major steps of the algorithm presented in Section 3.

5.2.1 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 all objects reachable from a remote call's parameters need to be traversed for that. The linear map that we need is just a data structure storing references to all such objects in the order that they were traversed. We get this data structure with a tiny change to the serialization code. The overhead is minuscule and only exists when call-by-copy-restore semantics have been chosen.

5.2.2 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 there is any.

5.2.3 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).

5.2.4 Optimizations. There are two optimizations that can be applied to an implementation like 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 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. NRMI applies the first optimization, while the second will be part of future work.

5.3. Implementation (Low-Level) and Performance Experiments

Although NRMI emphasizes usability, it also attempts to offer an efficient implementation. The main point of our experiments is to demonstrate that *NRMI is efficient enough for real-world use*. The implementation of NRMI is optimized and suffers only small overheads. The optimized NRMI for JDK 1.4 is about 20% slower than regular RMI for JDK 1.4. To put this number in perspective, this also means that NRMI for JDK 1.4 is about 20-30% faster than regular RMI for the previous version, JDK 1.3.

5.3.1 NRMI Low-Level Optimizations. In principle, the only significant overhead of call-by-copy-restore middleware over call-by-copy middleware is the cost of transferring the data back to the client after the remote routine

execution. In practice, middleware implementations suffer several overheads related to processing the data, so that processing time often becomes as significant as network transfer time. Java RMI has been particularly criticized for inefficiencies, as it is implemented as a high level layer on top of several general purpose (and thus slow) facilities—RMI often has to suffer the overheads of security checks, Java serialization, indirect access through mechanisms offered by the Java Virtual Machine, etc. NRMI has to suffer the same overheads to an even greater extent, since it has to perform an extra traversal and copying over object structures.

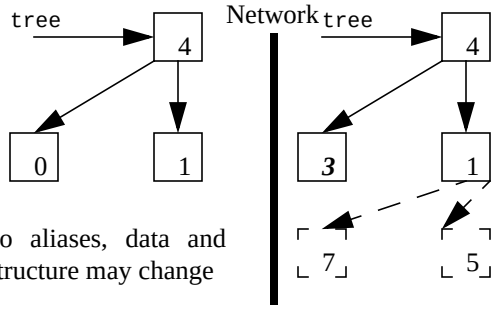
Currently, NRMI has two implementations: a "portable", high-level one and an "optimized" one. The *portable* implementation makes use of high-level features like Java reflection for traversing and copying object structures. Although NRMI is currently tied to Sun's JDK, the portable implementation works with both JDK 1.3 and JDK 1.4 on all supported platforms. The portability means some loss of performance: Java reflection is a very slow way to examine and update unknown objects. The NRMI implementation minimizes the overhead by caching reflection information aggressively. Additionally, the portable NRMI implementation uses native code for reading and updating object fields without suffering the penalty of a security check for every field. These two optimizations are sufficient for significantly reducing the NRMI overheads, even for the portable version. Our original implementation of NRMI was more than two times slower than the current portable one.

The *optimized* implementation of NRMI only works with JDK 1.4 and takes advantage of special features exported by the JVM in order to achieve better performance. The performance of regular Java RMI improved significantly between versions 1.3 and 1.4 of the JDK (as we show in our measurements). The main reason was the flattening of the layers of abstraction in the implementation. Specifically, object serialization was optimized through non-portable direct access to objects in memory through an "Unsafe" class exported by the Java Virtual Machine. The optimized version of NRMI also uses this facility to quickly inspect and change objects.

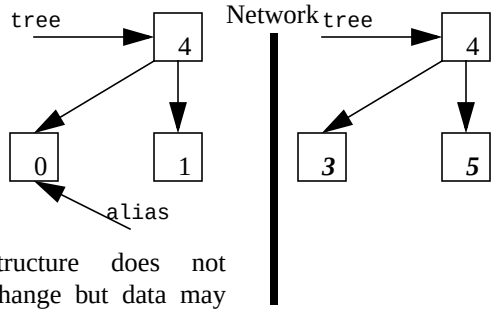
5.3.2 Description of Experiments. In order to see how our implementation of call-by-copy-restore measures up against the standard implementation of RMI, we created three micro-benchmarks. Each benchmark consists of a single randomly-generated binary tree parameter passed to a remote method. The remote method performs random changes to its input tree. *The invariant maintained is that all the changes are visible to the caller.* In other words, the resulting execution semantics is as if both the caller and the callee were executing within the same address space. With NRMI or distributed call-by-reference (through remote pointers, as in Figure 3) this is done automatically. For call-by-copy, the programmer needs to simulate it by hand.

We have considered three different scenarios of parameter use, listed in the order of difficulty of achieving the call-by-copy-restore semantics "by-hand" using the means provided by RMI.

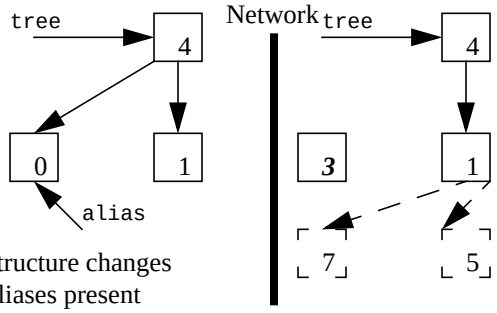
- In the first benchmark scenario, we assume that none of the objects reachable from the parameter is aliased by the client.



- In the second benchmark scenario, we allow aliases but assume that the structure of the tree stays the same (although the tree data may be modified by the remote method).



- In the third benchmark scenario, aliasing and modifications can be arbitrarily complex: tree nodes can be aliased on the client site and the tree structure can be changed by the remote call.



Consider how a programmer can replay the server changes on the client using regular Java RMI in each of the three cases. We assume that the programmer is fully aware of the server's behavior, as well as whether aliases exist on the client site.

- In the first case, the parameter just has to be returned as the return value of the remote method. Once the remote call completes, the reference pointing to the original data structure gets reassigned to point to the return value. This will work for any changes to both the data and structure of the tree. The only complication here is that the method might already have a return value. Resolving this problem would require defining a special return class type that would contain both the original return type and the parameter. Besides the code for this new return class type itself, some additional code has to be written to call its constructor, populate it with its constituent members on the callee site, and retrieve them when the call completes.
- In the second case, the client needs to reassign the aliases pointing to some nodes in the original tree to point to the corresponding nodes in the new tree. After this step is performed, the reference reassignment described in the previous benchmark can be used. If the programmer knows all the aliases, as well as where in the tree they point to (i.e., how to get to the aliased node from the tree root) then the aliases can be reassigned directly. If, however, the programmer only knows the aliases but not how to get to the aliased nodes, then a search needs to be performed before the update takes place. Both the original and the modified trees (that are now isomorphic) can be traversed simultaneously. Upon encountering each node, all aliases should be reassigned from pointing from the original tree to the modified one.
- In the third case, returning the changed structure alone is not very useful since the original and the modified trees are no longer isomorphic. To complicate matters further, the remote method invocation might make some changes to some of the tree nodes data that were aliased by the caller and then disconnect them from the tree structure. This way the modified data nodes might no longer be part of the tree structure. Obviously just returning the new tree is not enough. Emulating the call-by-copy-restore or call-by-reference semantics is particularly cumbersome in this case. The simplest way to do it is by having the remote method create a "shadow tree" of its tree parameter prior to making any changes to it. The "shadow tree" points to the original tree's data and serves as a reminder of the structure of the original tree. Then both the parameter tree and the "shadow tree" are returned to the caller. The "shadow tree" is isomorphic to the original parameter and can be used for simultaneous traversal and copying of aliases. After that the new tree is used for the reference reassignment operation as in the previous cases. Note that correct update is not possible without modifying both the server and the client.

For all benchmarks, the NRMI version of the distributed code is quite similar to the local version, with the exception of remote method lookup and declaring a class to be `Restorable`. Analogous changes have to be made in order to go from the local version to the distributed one that updates client data correctly using regular Java RMI. Several extra lines of code have to be added/modified in the latter case, however. For all three benchmarks, about 45 lines of code were needed in order to define return types. For the second and third benchmark scenario, an extra 16 lines of code were needed to perform the updating traversal. For the third benchmark scenario, about 35 more lines of code were needed for the "shadow tree".

5.3.3 Experimental Results. For each of these benchmarks, we measure the performance of call-by-copy (RMI), call-by-copy-restore (NRMI), and call-by-reference implemented using remote pointers (RMI). (Of course, NRMI can also be used just like regular RMI with identical performance. In this section when we talk of "NRMI" we mean "under call-by-copy-restore semantics".) For reference, we also provide three base line numbers by showing how long it takes to execute the same methods within the same JVM locally, on different JVMs through RMI but on the same physical machine, and on different machines but without caring to restore the changes to the client (i.e., only sending the tree to the server but not sending the changed tree back to the client). We show measurements for both implementations of NRMI and both JDK 1.3 and JDK 1.4. The environment consists of a SunBlade 1000 (two UltraSparc III 750MHz processors and 2GB of RAM) and a Sun Ultra 10 (UltraSparc II 440MHz) machines connected with a 100Mbps effective bandwidth network. This environment certainly does not unfairly benefit NRMI measurements—the network speed is on the high end of typical networks where high-level middleware is used and the machines have realistic speeds for this kind of processing. For faster machines and slower networks, the performance of NRMI would strictly improve relative to the baselines.

The results of our experiments are shown in Table 1-Table 6. All numbers are in milliseconds per remote call, rounded to the nearest millisecond. We ensured that all measured programs had been dynamically compiled by the JVM before the measurements. The local measurements of Table 1 are given for both the fast and the slow machines. The local measurements of Table 3 are from the dual processor SunBlade machine. (This allowed us to

Table 1: Baseline 1—Local Execution (processing overhead) on both the fast (750MHz) and the slow (440MHz) machine

Benchmark/ Tree Size	JDK 1.3				JDK 1.4			
	16	64	256	1024	16	64	256	1024
I	<1 / <1	<1 / 1	1 / 2	6 / 8	<1 / <1	<1 / 1	1 / 1	4 / 6
II	<1 / 1	1 / 1	4 / 5	15 / 20	<1 / 1	1 / 1	3 / 4	12 / 16
III	<1 / 1	1 / 2	5 / 6	19 / 24	<1 / 1	1 / 1	4 / 5	15 / 19

Table 2: Baseline 2—RMI Execution, without Restore (one-way traffic)

Benchmark/ Tree Size	JDK 1.3				JDK 1.4			
	16	64	256	1024	16	64	256	1024
I	3	7	18	65	2	4	9	33
II	3	7	21	74	3	4	12	41
III	3	8	22	79	3	5	12	44

Table 3: Baseline 3—RMI Execution with Restore on local machine (no network overhead)

Benchmark/ Tree Size	JDK 1.3				JDK 1.4			
	16	64	256	1024	16	64	256	1024
I	3	7	17	59	3	4	11	41
II	4	8	19	67	3	5	13	48
III	4	9	24	87	3	6	16	66

Table 4: RMI Execution with Restore (two-way traffic)

Benchmark/ Tree Size	JDK 1.3				JDK 1.4			
	16	64	256	1024	16	64	256	1024
I	5	11	29	102	4	6	18	68
II	5	12	32	112	4	7	21	77
III	6	13	38	143	4	9	27	106

Table 5: NRMI (Call-by-copy-restore). Both the portable and optimized implementation shown for JDK 1.4

Benchmark/ Tree Size	JDK 1.3				JDK 1.4			
	16	64	256	1024	16	64	256	1024
I	6	13	36	130	5 / 4	8 / 8	25 / 22	93 / 82
II	6	13	38	141	5 / 4	9 / 8	27 / 24	103 / 95
III	6	14	39	146	5 / 4	9 / 8	28 / 25	106 / 97

Table 6: Call-by-Reference with Remote References (RMI)

Benchmark/ Tree Size	JDK 1.3				JDK 1.4			
	16	64	256	1024	16	64	256	1024
I	41	50	87	-	44	48	124	-
II	35	50	85	-	49	53	95	-
III	113	123	164	-	131	131	228	-

avoid context switching and get a fair measurement. The numbers were significantly tainted on a uniprocessor machine.) The main observations from these measurements are as follows:

- The benchmarks have very low computation times—their execution consists mostly of middleware processing and data transmission.
- Java RMI in JDK 1.4 is significantly faster than RMI in JDK 1.3. The speedup is in the order of 50-60% for this experimental setting. The speedup will be lower for a network that is slower relative to the processor speeds.
- The results of Table 4 minus the corresponding results of Table 3 will only yield an upper bound for the raw data transmission time, because the Table 3 results were computed exclusively on the fast (750MHz) machine while the Table 4 results include computation on both the fast and the slow (440MHz) node. The difference between the raw data transmission time and the "Table 4-minus-Table 3" value can be as high as the difference between the computation times on the fast and slow machines, shown in Table 1. Even then, however, JDK 1.3 seems to perform much better when no network is involved than the corresponding difference in JDK 1.4. This leads us to conclude that probably RMI in JDK 1.4 uses the underlying OS/networking facilities much more efficiently than JDK 1.3 and this difference disappears when the two hosts are sharing memory. We independently corroborated the raw data transmission time shown in the tables by profiling the benchmarks and noting the amount of time they spent blocked for I/O. We found that the real transmission time for JDK 1.3 is much higher even for transmitting the exact same amount of data as 1.4.
- For benchmarks I and II, NRMI is quite efficient. Even the portable version is rarely more than 30% slower than the corresponding RMI version. The optimized implementation of NRMI is about 20% slower than RMI in JDK 1.4. This is certainly fast enough for use in real applications. For instance, the optimized implementation of NRMI for JDK 1.4 is 20-30% faster than regular RMI in JDK 1.3.
- For benchmark III, which is hard to simulate by hand with call-by-copy alone, the portable implementation of NRMI gets similar performance to regular RMI in all cases, while the optimized implementation is faster. The cause is *not* the processing time for restoring the values changed by the header. (In fact, we performed the same experiments ignoring the manual restoring of changes and got almost identical timings.) Instead, the reason is that the regular RMI version transfers more data: the "shadow tree" is a simple way to emulate the local semantics by hand, but stores more information than that of the NRMI linear map. (Specifically, it stores all the original structure of the tree instead of just pointers to all the reachable nodes.) The only alternative would be to compute a linear mapping to all reachable nodes on both sides, effectively imitating NRMI in user space.
- Call-by-reference implemented by remote pointers is extremely inefficient (as expected). Every access to parameter data by the remote method results in network traffic. Java RMI does not seem fit for this kind of communication at all—the memory consumption of the benchmarks grew uncontrollably. For the 1,024 node trees, the benchmarks took more than 600ms per case (repeated over 1,000 times) and in fact failed to complete as they exceeded the 1GB heap limit that we had set for our Java virtual machines. The reason for the memory leak is that the references back from the server to the client create distributed circular garbage. Since RMI only supports reference counting garbage collection, it cannot reclaim the garbage data.

The conclusion from our experiments is that NRMI is optimized enough for real use. NRMI (copy-restore) for JDK 1.4 is close to the optimized RMI in JDK 1.4 and faster than regular RMI (call-by-copy with results passed back) in JDK 1.3. Of course, with NRMI the programmer maintains the ability to use call-by-copy semantics when deemed necessary. When, however, a more natural programming model is desired, NRMI is without competition—the only alternative is the very slow call-by-reference through RMI remote pointers.

6. Related Work

Distributed computing has been the main focus of systems research in the past two decades. A large portion of the research community's efforts is aimed at improving the performance and usability aspects of distributed computing. Since the objective of NRMI lies mostly in usability but without sacrificing performance, there is a wealth of work that exhibits similar goals or methodologies to ours. Hence, we will be selective in our presentation and we will separate performance and usability related work.

6.1. Performance Improvement Work

There have been several efforts targeted at providing a more efficient implementation of the facilities offered by standard RMI [14]. Krishnaswamy et al. [6] achieve RMI performance improvements by replacing TCP with UDP and by utilizing object caching. Upon receiving a remote call, a remote object is transferred to and cached on the caller site. In order for the runtime to implement a consis-

tency protocol, the programmer must identify whether a remote method is read-only (e.g., will only read the object state) or not, by including the throwing of "read" or "write" exceptions. That is, instead of transferring the data to a read-only remote method, the server object is moved to the data instead, which results in better performance in some cases.

There are several systems that improve the performance of RMI by using a more efficient serialization mechanism. KaRMI [8] uses a serialization implementation based on explicit routines for writing and reading instance variables along with more efficient buffer management. Maassen et al.'s work [7] takes an alternative approach by using native code compilation to support compile and run time generation of marshalling code.

It is interesting to observe that most of the optimizations aimed at improving the performance of the standard RMI and call-by-copy can be successfully applied to NRMI and call-by-copy-restore. Furthermore, such optimizations would be even more beneficial to NRMI due to its heavier use of serialization and networking.

6.2. Usability Improvement Work

Thiruvathukal et al. [20] propose an alternative approach to implementing a remote procedure call mechanism call for Java based on reflection. The proposed approach employs the reflective capabilities of the Java languages to invoke methods remotely. This simplifies the programming model since a class does not have to be declared remote for its instances to receive remote calls.

While CORBA does not currently support object serialization, the OMG is reviewing the possibilities of making such support available in some future version of IIOP [7]. Once object serialization becomes available, both call-by-copy and call-by-copy-restore can be implemented enabling [in] and [in out] parameters passing semantics for objects.

Distributed Shared Memory (DSM) systems are the main way employed in the research literature to making distributed computing easier. Traditional DSM approaches create the illusion of a shared address space, when the data are really distributed across different machines. Example DSM systems include Munin [3], Orca [2], and, in the Java world, CJVM [1], and Java/DSM [23]. DSM systems can be viewed as sophisticated implementations of call-by-reference semantics, to be contrasted with the naive "remote pointer" approach shown in Figure 3. Nevertheless, the focus of DSM systems is very different than that of middleware. DSMs are used when distributed computing is a means to achieve parallelism. Thus, they have concentrated on providing correct and efficient semantics for multi-threaded execution. To achieve performance, DSM systems create complex memory consistency models and require the programmer to implicitly specify the sharing properties of data. In practice, the applicability of DSMs has been restricted to high-performance parallel applications, mainly in a research setting. In contrast, NRMI attempts to support natural semantics to straightforward middleware, which is always under the control of the programmer. NRMI (and all other middleware) do not try to support "distribution for parallelism" but instead facilitate distributed computing in the case where an application's data and input are naturally far away from the computation that needs them.

A special kind of tools that attempt to bridge the gap between DSMs and middleware are *automatic partitioning tools*. Such tools split centralized programs into several distinct parts that can run on different network sites. Thus, automatic partitioning systems try to offer DSM-like behavior but with more ease of use and compatibility: Automatically partitioned applications run on existing infrastructure (e.g., DCOM or regular unmodified JVMs) but relieve the programmer from the burden of dealing with the idiosyncrasies of various middleware mechanisms. At the same time, this reduces the field of application to programs where locality patterns are very clear cut—otherwise performance can suffer greatly. In the Java world, the J-Orchestra [19], Addistant [18] and Pangaea [12][13] systems can be classified as automatic partitioning tools.

The JavaParty system [5][11] works much like an automatic partitioning tool, but gives a little more programmatic control to the user. JavaParty is designed to ease distributed cluster programming in Java. It extends the Java language with the keyword `remote` to mark those classes the can be called remotely. The JavaParty compiler then generates the required RMI code to enable remote access. Compared to NRMI, JavaParty is much closer to a DSM system, as it incurs similar overheads and employs similar mechanisms for exploiting locality.

Doorastha [4] represents another piece of work on making distributed programming more natural. Doorastha allows the user to annotate a centralized program to turn it into a distributed application. Although Doorastha allows fine-grained control without needing to write complex serialization routines, the choice of remote calling semantics is limited to call-by-copy and call-by-reference implemented through RMI remote pointers or object mobility. Call-by-copy-restore can be introduced orthogonally in a framework like Doorastha. In practice, we expect that call-by-copy-restore will often be sufficient instead of the costlier, DSM-like call-by-reference semantics.

Finally, we should mention that approaches that hide the fact that a network is present have often been criticized (e.g., see the well-known Waldo et al. "manifesto" on the

subject [22]). The main point of criticism has been that distributed systems fundamentally differ from centralized systems because of the possibility of partial failure, which needs to be handled differently for each application. The "network transparency" offered by NRMI does not violate this principle in any way. Just like RMI, NRMI remote methods throw remote exceptions that the programmer is responsible for catching. Thus, programmers are always aware of the network's existence, but with NRMI they often do not need to program differently, except to concentrate on the important parts of distributed computing, i.e., handling partial failure.

7. Conclusions

Distributed computing has moved from an era of "distribution for parallelism" to an era of "data-driven distribution": the data sources of an application are naturally remote to each other or to the computation. In this setting, call-by-copy-restore is a very useful middleware semantics, as it closely approximates local execution. In this paper we describe the implementation and benefits of call-by-copy-restore middleware for arbitrary linked data structures. Our ideas are concretely implemented in NRMI: an efficient Java middleware mechanism that supports call-by-copy-restore semantics in addition to standard call-by-copy semantics. We believe that NRMI is a valuable tool for Java distributed programmers and that the same ideas can be applied to middleware design and implementation for other languages.

Acknowledgments

This work has been supported by NSF through an ITR award, by the Yamacraw Foundation, and by DARPA/ITO under the PCES program. Equipment was provided by Sun Corporation through an Academic Equipment Grant. Doug Lea and Cristina Videira Lopes read early drafts of this paper and offered valuable comments.

8. References

[1] Yariv Aridor, Michael Factor, and Avi Teperman, "CJVM: a Single System Image of a JVM on a Cluster", in Proc. *ICPP'99*.

[2] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Ruhl, and M. Frans Kaashoek, "Performance Evaluation of the Orca Shared-Object System", *ACM Trans. on Computer Systems*, 16(1):1-40, February 1998.

[3] John B. Carter, John K. Bennett, and Willy Zwaenepoel, "Implementation and performance of Munin", *Proc. 13th ACM Symposium on Operating Systems Principles*, pp. 152-164, October 1991.

[4] Markus Dahm, "Doorastha—a step towards distribution transparency", *JIT*, 2000. See `http://www.inf.fu-berlin.de/~dahm/doorastha/` .

[5] Bernhard Haumacher, Jürgen Reuter, Michael Philippsen, "JavaParty: A distributed companion to Java", `http://wwwipd.ira.uka.de/JavaParty/`

[6] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, Mustaque Ahamad, "Efficient Implementations of Java Remote Method Invocation (RMI)", *Proc. of Usenix Conference on Object-Oriented Technologies and Systems (COOTS'98)*, April 1998.

[7] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, Aske Plaat, "An Efficient Implementation of Java's Remote Method Invocation", *Proc. of ACM Symposium on Principles and Practice of Parallel Programming, Atlanta, GA May 1999.*

[8] Christian Nester, Michael Phillipsen, and Bernhard Haumacher, "A More Efficient RMI for Java", in Proc. *ACM Java Grande Conference*, 1999.

[9] OMG. Objects by Value Specification, `http://www.omg.org/cgi-bin/doc?orbos/98-01-18.pdf`, January 1998.

[10] The Open Group. DCE 1.1 RPC Specification, 1997. `http://www.opengroup.org/onlinepubs/009629399/`

[11] Michael Philippsen and Matthias Zenger, "JavaParty - Transparent Remote Objects in Java", *Concurrency: Practice and Experience*, 9(11):1125-1242, 1997.

[12] Andre Spiegel, "Pangaea: An Automatic Distribution Front-End for Java", 4th *IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99)*, San Juan, Puerto Rico, April 1999.

[13] Andre Spiegel, "Automatic Distribution in Pangaea", *CBS 2000*, Berlin, April 2000. See also `http://www.inf.fu-berlin.de/~spiegel/pangaea/`

[14] Sun Microsystems, Java Object Serialization Specification, `ftp://ftp.java.sun.com/docs/j2se1.4/serial-spec.ps`, 2001.

[15] Sun Microsystems, Remote Method Invocation Specification, `http://java.sun.com/products/jdk/rmi/`, 1997.

[16] Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice-Hall, 1995.

[17] Andrew S. Tanenbaum and Maarten van Steen, *Distributed Systems: Principles and Paradigms*, Prentice-Hall, 2002.

[18] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano, "A Bytecode Translator for Distributed Execution of 'Legacy' Java Software", *European Conference on Object-Oriented Programming (ECOOP)*, Budapest, Hungary, June 2001.

[19] Eli Tilevich and Yannis Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning", *European Conference on Object-Oriented Programming (ECOOP)*, Malaga, Spain, June 2002.

[20] George K. Thiruvathukal, Lovely S. Thomas, and Andy T. Korczynski. "Reflective remote method invocation.", *Concurrency: Practice and Experience*, 10(11-13):911-926, September-November 1998.

[21] Unknown, "Distributed Computing Systems course notes", `http://www.cs.wpi.edu/~cs4513/b01/week3-comm/week3-comm.html`

[22] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, "A note on distributed computing", Technical Report, Sun Microsystems Laboratories, SMLI TR-94-29, Nov. 1994.

[23] Weimin Yu, and Alan Cox, "Java/DSM: A Platform for Heterogeneous Computing", *Concurrency: Practice and Experience*, 9(11):1213-1224, 1997.