NRMI: Natural and Efficient Middleware

Eli Tilevich, *Member, IEEE,* and Yannis Smaragdakis, *Senior Member, IEEE*

***Abstract*— We present 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. Call-by-copy-restore offers a more natural programming model for distributed systems than traditional call-by-copy middleware, enabling remote calls to behave much like local calls. We discuss in depth the effects of calling semantics for middleware, describe when and why NRMI is more convenient to use than standard middleware, and present three implementations of NRMI in distinct settings, showing the generality of the approach.**

***Index Terms*— Middleware, RPC, Java, call-by-copy-restore, programming model.**

I. 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, current call-by-copy-restore middleware cannot 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) Tanenbaum and van Steen "Distributed Systems" textbook [2] 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 article 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. Specifically, 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.

Our algorithm has been implemented in the form of Natural Remote Method Invocation (NRMI): a middleware facility with three different implementations. The first is a drop-in replacement for Java RMI; the second is in the context of the J2EE platform; and the third introduces call-by-copy-restore 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. The resulting advantage is that NRMI offers a much more natural distributed programming model than standard Java RMI: in most cases, programming with NRMI is identical to non-distributed Java programming. In fact, call-by-copy-restore is 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, NRMI often offers behavior practically indistinguishable from local calls.

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 IV-B.

In summary, this article makes the following contributions:

- We give 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).
- We present an algorithm for implementing call-by-copy-restore middleware simply and efficiently. This is the first algorithm to implement full call-by-copy-restore for arbitrary linked data structures.
- We make a case for the advantages of using call-by-copy-restore semantics in actual middleware. We argue that call-by-copy-restore results in a more natural programming model that significantly simplifies programming tasks when data passed to a remote call are reachable through multiple pointers. This simplicity does not sacrifice the efficiency of the remote procedure call mechanism.
- We demonstrate an applied result in the form of three concrete implementations of NRMI. NRMI is a mature and

E. Tilevich is with the Department of Computer Science, Virginia Tech.
Y. Smaragdakis is with the Department of Computer and Information Science, University of Oregon.
This article is an extended version of [1].

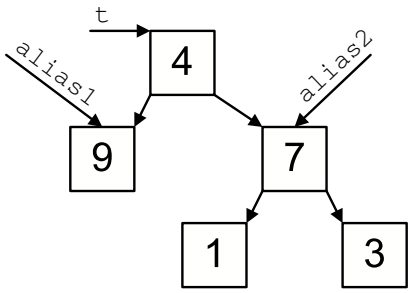

Fig. 1. A tree data structure and two aliasing references to its internal nodes.

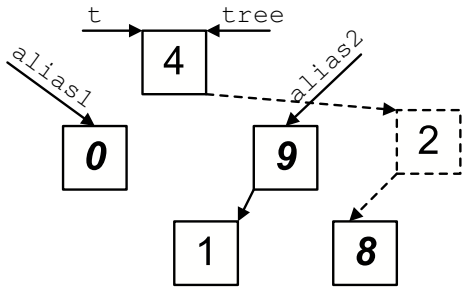

Fig. 2. A local call can affect all reachable data.

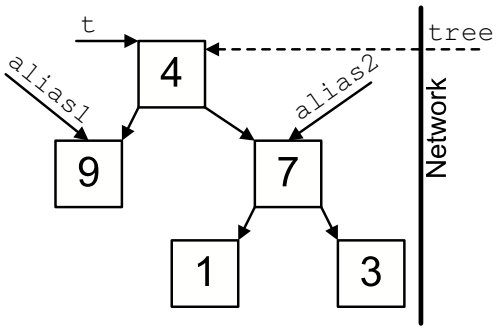

Fig. 3. Call-by-reference semantics can be maintained with remote references.

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 applications 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.

II. 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 article. 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 that some of the subtrees are also pointed to by non-tree pointers (aka *aliases*). Figure1 shows an instance of such a tree.

When 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 alterTree(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 shows the results on the data structure after performing a call `alterTree(t)` locally. (New number values shown in bold and italic, new nodes and references are dashed. Null references are not shown.)

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 article, 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 `alterTree(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 `alterTree` 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 3. Most object-oriented middleware 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, this solution is extremely inefficient. It means that every pointer dereference has to generate network traffic. There-

fore, 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 callers original value" [2]. 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 `alterTree` 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 IV, 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 presents several complications. Our example function `alterTree` 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).

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

1) Create a linear map of all objects reachable from the remote call reference argument. 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 argument, as it is itself 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 V) back to the client site. This will copy back all the "interesting" objects, even if they have become unreachable from the original remote call argument data.
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 existed 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 in-place 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.

Fig. 4. NRMI Algorithm

III. Supporting Copy-Restore

Having introduced the complications of copy-restore middleware, we now discuss an algorithm that addresses them. The algorithm appears in pseudo-code in Figure 4 and is illustrated on our running example in Figures 5 to 8.

The above algorithm reproduces the modifications introduced by the server routine on the client data structures. The intuition behind the algorithm is that correct call-by-copy-restore behavior requires restoring *all changes to data that are reachable (after the execution of the remote call) from any data that used to be reachable (before the execution of the remote call) from any of the arguments of the remote call*. Thus, 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: although there is an overhead in setting up the argument data, the remote code itself proceeds at full speed. In particular, the algorithm 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.

IV. Discussion

A. 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

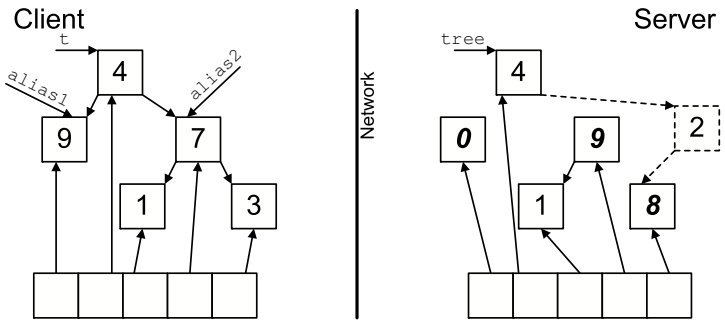

Fig. 5. State after steps 1 and 2 of the algorithm. Remote procedure `alterTree` has performed modifications to the server version of the data.

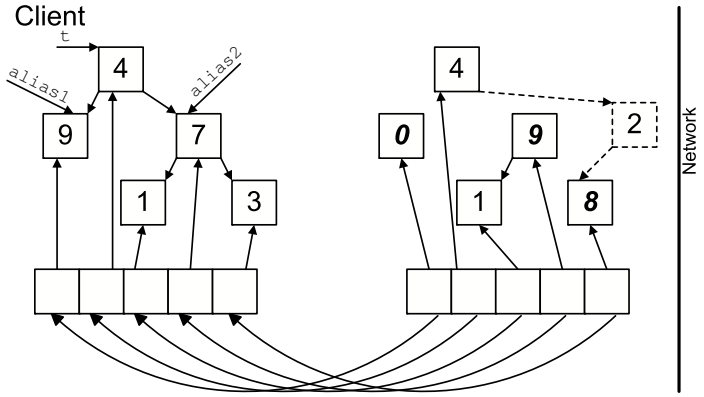

Fig. 6. State after steps 3 and 4 of the algorithm. The modified objects (even the ones no longer reachable through `tree`) are copied back to the client. The two linear representations are "matched"—i.e., used to create a map from modified to original versions of old objects.

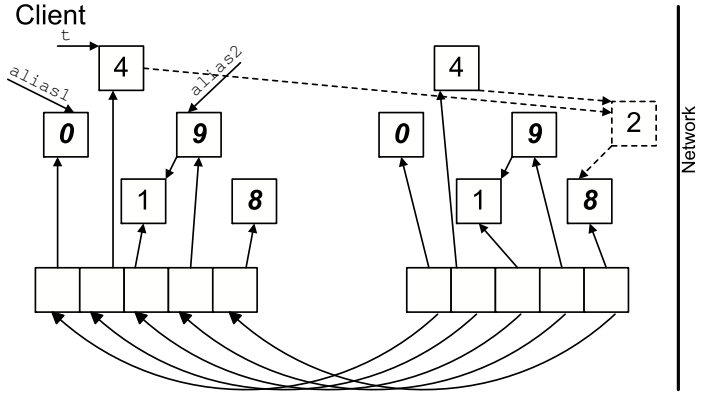

Fig. 7. State after step 5 of the algorithm. All original versions of old objects are updated to reflect the modified versions.

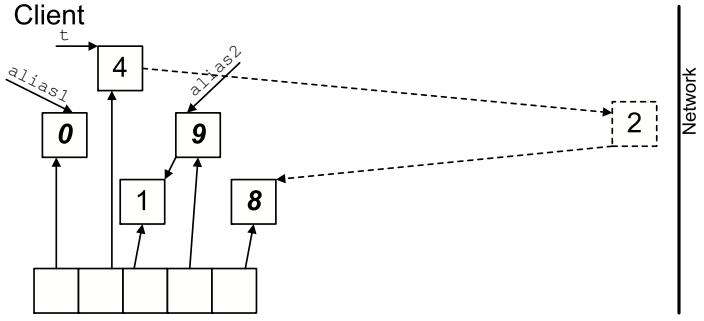

Fig. 8. State after step 6 of the algorithm. All new objects are updated to point to the original versions of old objects instead of their modified versions. All modified old objects and their linear representation can now be deallocated. The result is identical to Figure 2.

on the client site, call-by-copy-restore closely approximates local execution. In fact, we 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. Otherwise, if the remote routine keeps references to its input data across executions, then the two copies of the data (on the client and server) can become inconsistent after the "restore" action: call-by-copy-restore middleware only "synchronizes" the client and server data at the point of a remote call return. In this case, call-by-copy-restore will behave differently from call-by-reference, since the latter only shares one copy of the data between client and server. Interestingly, statelessness is a desirable (for many even indispensable) property for distributed services due to fault tolerance considerations. Thus, a call-by-copy-restore semantics achieves *network transparency* for the important case of stateless routines: the routine can be executed either locally or remotely with identical results.

The above discussion only considers single-threaded programs. In the case of a multi-threaded client, call-by-copy-restore middleware does not preserve network transparency. Consider a client process with two threads: one passing data to a remote routine, while another is modifying the same data. The remote routine acts as a potential mutator of all data reachable by its parameters. Yet when the changes are replayed on the client site, they will not be done by following the synchronization protocol of the data, and, thus, may conflict with the concurrent local changes. For instance, if the data passed to the remote routine is a list with each node protected by its own mutex, the remote routine will make changes without contention: it is the only mutator of these data on the server site. Yet when the changes are replayed on the client site, they will be made all at once by the middleware system and not through code that takes the appropriate synchronization actions. Therefore, the changes may conflict with those of other client-side threads. The programmer needs to be aware that the call is remote and that a call-by-copy-restore semantics is used. For instance, remote calls may need to execute in mutual exclusion with calls that read/write the same data. If the order of updating matters, call-by-copy-restore probably can not be used at all: the programmer needs to write code by hand to perform 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 this 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 [2]–[4].

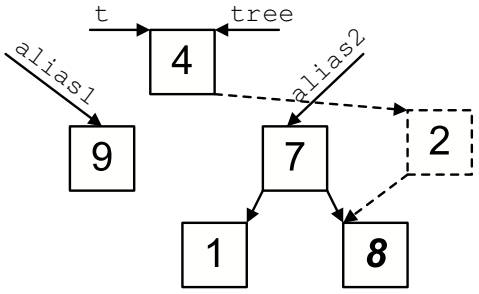

Fig. 9. Under DCE RPC, changes to data that became unreachable from `t` are not restored on the client.

B. DCE RPC

The DCE RPC specification [5] 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 II. 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. Figure 9 shows the actual results for DCE RPC.

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

Compared to call-by-copy, call-by-copy-restore semantics offers better usability, since it simulates the local execution seman-

[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.*"

tics very closely, as discussed in Section IV-A. Clearly, call-by-copy-restore semantics can be achieved by using call-by-copy and adding application-specific code to register and re-perform any updates necessary. Nevertheless, taking this approach 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 VI-C).
- The programmer cannot perform the updates without full knowledge of what changes the server code made. 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 discussed in SectionII, a call-by-copy-restore semantics is most valuable in the presence of aliased data. Aliasing occurs as a result of several common implementation techniques in mainstream programming languages. All of these techniques produce code that is more convenient to write using call-by-copy-restore middleware than using call-by-copy middleware. Specific examples include:

- *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 customers record, through a reference from the daily tax record object, and so forth. 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 (i.e., 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 customers address from a central database), in much the same way as they would be updated if the call were local.
- *Common GUI patterns such as 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 can be the result of a remote call. A variant of this pattern occurs when GUI elements (e.g., menus, toolbars) 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, and so forth or that the data may need to be modified programmatically with the changes reflected in the menu or toolbar.

  As an illustration, 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, and so on. The NRMI distributed version code has only two tiny changes compared to local code: a single class needs to implement `java.rmi.Restorable` (the NRMI marker interface, discussed in detail later) 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. Figures 10-12 show the different versions of the code context containing the key remote call of this application. The complexity of the special-purpose RMI update code is evident in Figure 12.

```
TranslationService ts =
  new TranslationService();
Translatable trans =
  new Translatable(srcLang, destLang, _tokens);
ts.translate (trans);
srcLang = destLang;
redraw ();
```

Fig. 10. Code Fragment of Local Version of Translation Code

```
String url = host + "/transl_service";
TranslationServiceInterface t =
  (TranslationServiceInterface)lookup (url);
Translatable trans =
  new Translatable(srcLang, destLang, _tokens);
t.translate (trans);
srcLang = destLang;
redraw ();
```

Fig. 11. Code Fragment of NRMI Version of Translation Code

```
String url = host + "/transl_service";
TranslationServiceInterface t =
  (TranslationServiceInterface)lookup (url);
Translatable trans =
  new Translatable(srcLang, destLang, _tokens);
Vector temp = t.translate (trans);
for (int i = 0; i < temp.size (); ++i) {
  Token newToken = (Token)temp.elementAt (i);
  String str = newToken.getString ();
  int j = 0;
  Token token = null;
  for ( ; j < _tokens.size(); j++) {
    token = (Token)_tokens.elementAt (j);
    if (token.getString().equals(str)) break;
  }//for j
  if (j < _tokens.size())
    token.setString(newToken.getTranslation());
}//for i
srcLang = destLang;
redraw ();
```

Fig. 12. Code Fragment of RMI Version of Translation Code

V. NRMI Implementations

NRMI currently has three implementations, each applicable to different programming environments and scenarios. Our first implementation is in the form of a full, drop-in replacement for Java RMI. This demonstrates how a mainstream 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 [6] to introduce NRMI as a pair of client/server interceptors. Another introduces NRMI by retrofitting the byte-codes 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.

A. A Drop-in Replacement of Java RMI

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. 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 NRMI implementation introduces a marker interface `java.rmi.Restorable` which allows the programmer to choose the by-copy-restore semantics on a per-type basis. For example:

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

`Restorable` extends `Serializable`, reflecting the fact that call-by-copy-restore is 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.

In the case of JDK classes, which cannot be modified, `Restorable` can be implemented by a subclass:

```
//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, where a class that implements `Restorable` serves as a proxy, forwarding calls to a JDK class.

Declaring a class to implement `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 discuss in Section VI-B.

In order to make NRMI easily applicable to existing types (e.g., arrays) that cannot be changed to implement `Restorable`, we adopted the policy that a serializable object is passed by-copy-restore, if it is referenced by an object that implements `Restorable`. Thus, if a parameter is of a "restorable" type, everything reachable from it will be passed by-copy-restore, if in regular Java RMI it would have been passed by-copy.

2) Implementation Insights: We next discuss how our implementation handles each of the major steps of the algorithm presented in Section III.

Creating a linear map: The 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 there is any.

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 is possible future work.

B. NRMI in the J2EE Application Server Environment

A J2EE [7] 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.

We have implemented NRMI in the application server environment of the JBoss open-source J2EE server, taking advantage of its extensible architecture [6]. JBoss is an extensible, open-ended, and dynamically-reconfigurable server. It employs the Interceptor pattern [8] (a common extensibility-enhancing mechanism in complex software systems) to enable transparent addition and automatic triggering of services. Informally, a JBoss interceptor is a piece of functionality that gets inserted into the client-server communication path. Both the clients requests to the server and the servers replies can be intercepted. 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 uses them to implement a large part 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 marker interface for copy-restore parameters. We introduced instead a new XDoclet [9] 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 annotation.

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

Note that, in this implementation, it is not possible to let the programmer 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.

To implement NRMI in JBoss, we had to create special interceptor classes for the client and the server portions of the code. The interceptors are invoked only for those methods specified as having call-by-copy-restore semantics. Both interceptors are implemented in about 100 lines of Java code. (This number excludes the actual NRMI algorithm implementation, which is another 700 lines of code.) This is a data point arguing that call-by-copy-restore can be implemented very simply even in a commercial quality middleware platform.

C. 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 that supports 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, our J-Orchestra automatic partitioning system [10], 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 we have argued earlier, access to a remote object through a remote reference incurs heavy 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 [11], [12]. 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. A small runtime system, consisting of code that implements the NRMI algorithm, is bundled with the original program, and the target classes are rewritten to invoke the NRMI functionality appropriately during remote calls.

Specifically, we offer a GUI-enabled tool called NRMIzer that takes as input 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. Under Suns 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 implemented 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.

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 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`.

```
public float foo(int i, Ref r)
  throws RemoteException{...}
```

If we want to pass the `Ref` parameter by copy-restore, the transformations performed on the stub code are as follows:

```
//change foo as follows (slightly simplified)
public float foo (int i, Ref r)
  throws RemoteException
{
  Object[] linearMap= NRMI.computeLinearMap(r);
  NRMIReturn ret = foo__nrmi (i, r);
                   //invoke foo__nrmi remotely
  // NRMIReturn encapsulates both linear maps
```

```
  // and the return value of foo
  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.

VI. Performance and Case Studies

Before presenting the results of NRMI performance experiments, we describe the performance optimizations that we applied to the RMI-replacement implementation of NRMI.

A. 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, and so forth. NRMI has to suffer the same and even higher overheads, since it has to perform an extra traversal and copying over object structures.

Our implementation of NRMI as a full replacement of Java RMI has two versions: a “portable”, high-level one and an “optimized” one. The *portable* version makes use of high-level features such as Java reflection for traversing and copying object structures. Although NRMI is currently tied to Suns JDK, the portable version works with JDK 1.3, 1.4, and 1.5 on all supported platforms and should be easy to port to other implementations. The portability means loss of performance: Java reflection is a slow way to examine and update unknown objects. Nevertheless, our implementation minimizes the overhead by caching reflection information aggressively. Additionally, the portable version uses JNI native code for reading and updating object fields without suffering the penalty of a security check for every field. These two optimizations give a ¿200% speedup to the portable version, but still do not achieve the optimized versions performance.

The *optimized* version of NRMI only works with Suns JDK 1.4 and 1.5 to take advantage of special low-level 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. 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 VM. The optimized version of NRMI also uses this facility to quickly inspect and change objects. We use the optimized NRMI version (also supporting the optimization discussed in Section V-A) in our experiments.

B. 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 multiple micro-benchmarks. Our benchmarks test NRMI with arrays, binary trees holding small objects, and binary trees holding arrays of multiple objects in each node. All data and their linking structure are randomly generated and passed to a remote method. The remote method performs random changes to its input data. We have considered different scenarios of parameter use for each data structure:

- For arrays, we consider the case of updating their contents, without changing the reference to the array itself (labeled “Array Benchmark 1 (Keep Reference)” in our plots) and the case of updating the entire array on the server (labeled “Array Benchmark 2 (Reset Reference)”).
- For binary trees of simple objects, as well as for binary trees of medium-size arrays, we used three scenarios (labeled “...Case 1 (No aliases, data and structure changes)”, “...Case 2 (Structure doesn’t change, but data does)”, and “...Case 3 (Structure changes, aliases present)”, respectively) of increasing complexity. In the first scenario, there is no aliasing on the client site of data passed as parameters to the remote call. In the second scenario, the remote call makes changes to data aliased on the client, but the linking structure of the data (i.e., the shape of the tree) does not change. In the third scenario, both the data and the tree structure change randomly and client aliases need to see these changes.

The invariant maintained is that all 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, we need to simulate this behavior by hand. We made a best-effort attempt to emulate what a programmer would actually do. We assumed that the programmer has full knowledge of the aliases on the client site, but no knowledge of what (random) changes were performed on the server site. (To be more exact, in scenario 2 for the binary tree benchmarks, the code assumes that the *location* of changes is known. In scenario 3, this is impossible, however, as the structure of the tree itself has changed.) Although we believe that our call-by-copy RMI code reflects what a real programmer would do, it may be viewed as pessimistic: it includes overhead for establishing general data structures that the server code uses to register its performed changes and the client code uses to restore them. In practice, the programmer may be able to do better by exploiting knowledge about the server-induced changes. In all our benchmarks, we also show the cost of RMI without any restoring logic, to establish a lower bound of the best results achievable.

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`. The same changes have to be made in the regular Java RMI call-by-copy version. Several additional lines of code have to be added/modified in the RMI call-by-copy case, however. For

instance, in the case of binary trees, all three benchmark scenarios required about 45 extra lines of code in order to define return types. For scenarios 2 and 3, an extra 16 lines of code were needed to perform the updating traversal. For scenario 3, about 35 more lines of code were needed for registering the changed data.

C. Experimental Results

We measured 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".) Our binary trees hold small objects (13 words in serialized form). Arrays have 100 slots and store integers. We performed our measurements with Sun's JDK (build 1.5.0_09-b03). Our environment consists of two Pentium D 3.GHz (dual core) machines, each with 2GB of RAM on a 100Mbps network. We tried to ensure (by "warming" the JVM) that all measured programs had been dynamically compiled before the measurements. To establish the baseline communication startup cost, we measured the time taken for a remote call to a routine with no arguments (`foo()`), as well as a routine being passed a `null` object reference (`foo(null)`). Both calls took about 0.2ms on average (0.205ms and 0.207ms, respectively), which is a small percentage of the execution cost of most of our benchmarks.

The results of our experiments are shown in Figure 13. The graphs show log-log plots of average execution time (over 10,000 iterations) of each benchmark. Across all benchmarks, NRMI performed quite similarly to the hand-written RMI call-by-copy version, while dramatically outperforming the RMI call-by-reference version.

The results hold over data structures with quite different behaviors and balances of communication and computation. As can be seen, serializing arrays and transferring them over the network is quite efficient—a relatively large percentage of the time is spent in communication overheads (compare the baseline `foo(null)` cost, above). In contrast, we see a much higher serialization cost for linked data structures. For all benchmarks, NRMI performance is quite close to the call-by-copy RMI version with hand-coded restore code. For simpler benchmark scenarios, the hand-coded solution performs up to 35% better, although typically the difference is in the 10-15% range. For more complex scenarios (i.e., "keep reference" for arrays, or scenario 3 of both binary trees and binary-tree-with-arrays) NRMI performs practically identically to the hand-coded RMI solution—the plot lines are almost completely overlapping. The one-way (no restore) RMI baseline is consistently at roughly half the execution time, indicating that the main overheads are due to the restore process. Finally, RMI with remote pointers (call-by-reference) is consistently orders of magnitude slower than any other solution. In fact, RMI with remote pointers failed to complete for large inputs of the linked structures examples, because it exhausted the heap. The reason for the memory leak is that RMI only has reference counting for distributed garbage collection and, hence, cannot reclaim data in cyclical reference patterns over the network.

Overall, our experiments show that NRMI is the only alternative efficient enough for real use that does not burden the programmer with writing specialized "restore" code for server-modified data. NRMI performs close (from 5% faster to about 35% slower) to regular (call-by-copy) RMI while offering a more natural programming model, which eliminates conceptual complexity and saves many lines of code per remote call. The only alternative that achieves the same code simplicity is the much less efficient call-by-reference through RMI remote pointers.

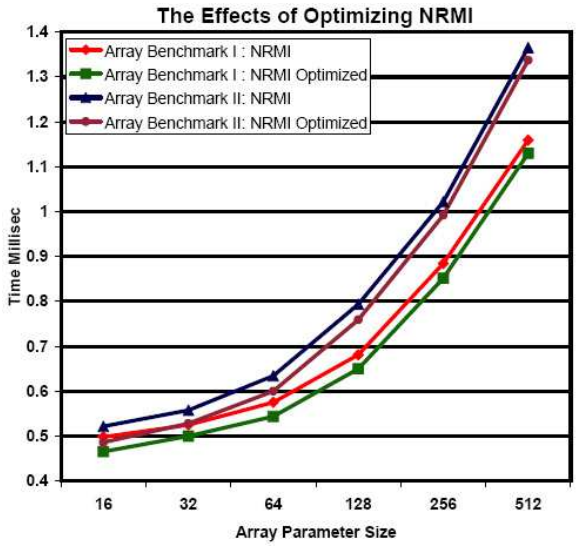

Fig. 14. Effect of recomputing linear map instead of serializing it.

Additionally, we measured the impact of the optimization described in Section V-A: re-creating the linear map during de-serialization of the remote call arguments, instead of passing it over the network. This takes advantage of fast CPUs to minimize network traffic. We show the effect of the optimization on our array benchmark in Figure 14. As can be seen, the optimized version of NRMI is about 5% faster than the unoptimized version. Thus, this optimization does not yield tremendous benefit, but helps remove some of the inherent overhead of NRMI over a plain RMI solution, especially for smaller inputs.

VII. Related Work

A. Performance and Scalability Improvement Work

Several efforts aim at providing a more efficient implementation of the facilities offered by standard RMI [13]. Krishnaswamy et al. [14] 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 consistency 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.

Several systems improve the performance of RMI by using a more efficient serialization mechanism. KaRMI [15] 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 [16], [17] 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

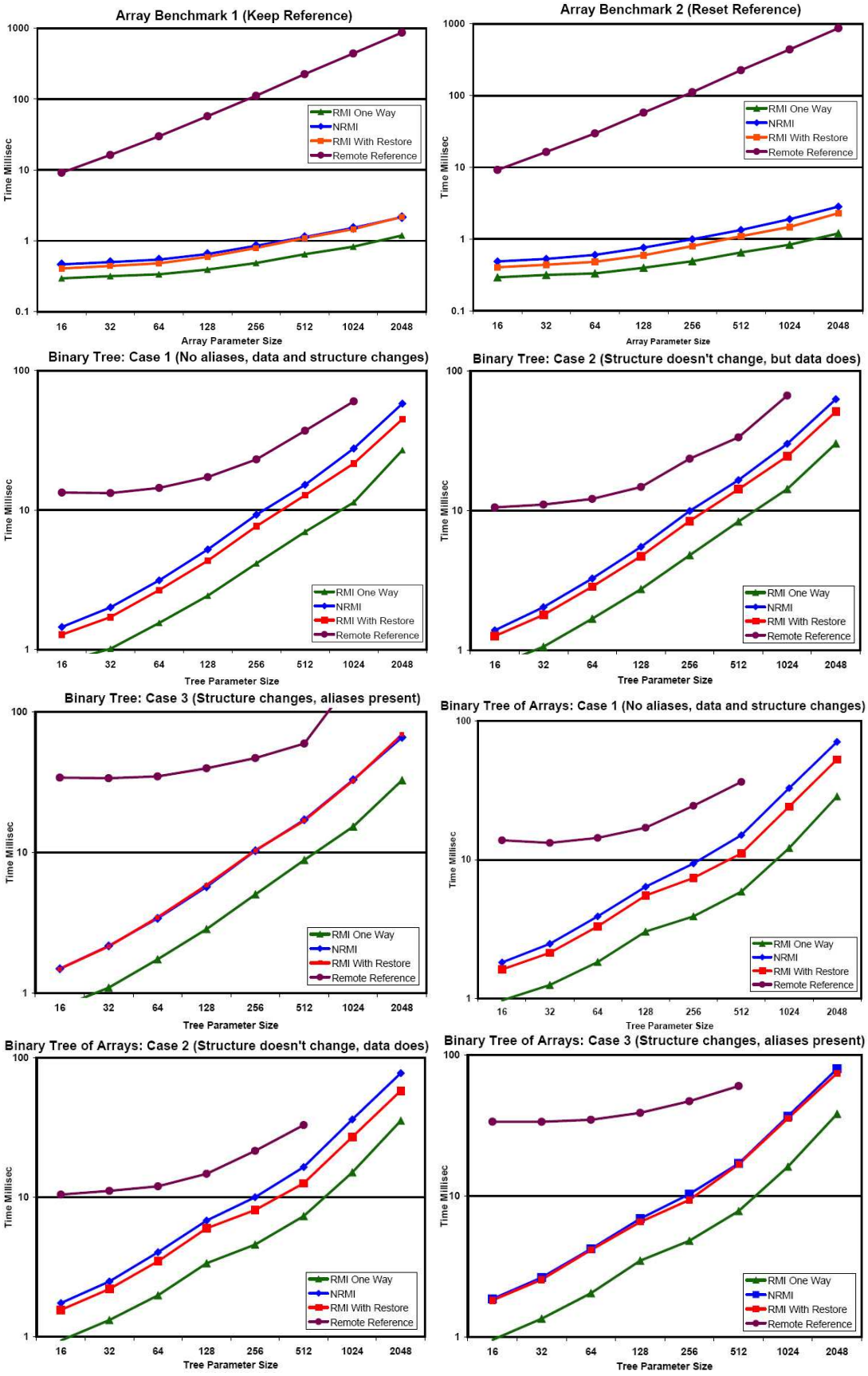

Fig. 13. Performance Comparison: RMI call-by-copy, NRMI call-by-copy-restore, RMI call-by-reference (remote pointers).

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.

B. Usability Improvement Work

Thiruvathukal et al. [18] propose an alternative approach to implementing a remote procedure call mechanism call for Java based on reflection. The approach employs the reflective capabilities of the Java language 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 has been reviewing the possibilities of making such support available in some future version of IIOP [19]. If object serialization becomes standardized, both call-by-copy and call-by-copy-restore can be implemented enabling [in] and [in out] parameter passing semantics for objects.

The systems research literature identifies Distributed Shared Memory (DSM) systems as a primary research direction aimed at 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 [20], Orca [21], and, in the Java world, cJVM [22], DISK [23], and Java/DSM [24]. 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. That is, NRMI does not attempt to offer distribution transparency, but instead achieves a more natural programming model that is still explicit. NRMI (and all other middleware) do not try to support "distribution for parallelism" but instead facilitate distributed computing in the case where an applications 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 distinct parts that can run on different network sites. Thus, automatic partitioning systems try to offer DSM-like behavior but with emphasis on automation and not performance: 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 [10], Addistant [25] and Pangaea [26] systems can be classified as automatic partitioning tools.

The JavaParty system [27], [28] 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 that 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 [29] 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 [30]). 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. Identically to regular 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 such as handling partial failure.

VIII. Conclusion

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 article we described the implementation and benefits of call-by-copy-restore middleware for arbitrary linked data structures. 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. We also presented detailed performance measurements of our drop-in RMI replacement implementation, proving that NRMI can be implemented efficiently enough for real world use. 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 was supported by the National Science Foundation under Grant No. CCR-0238289.

References

[1] E. Tilevich and Y. Smaragdakis, "NRMI: Natural and efficient middleware," in *International Conference on Distributed Computer Systems (ICDCS)*, 2003, pp. 252–261.

[2] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2002.
[3] A. S. Tanenbaum, *Distributed Operating Systems*. Prentice-Hall, 1995.
[4] Unknown, "Distributed computing systems course notes," http://www.cs.wpi.edu/∼cs4513/b01/week3-comm/week3-comm.html, accessed Apr. 2007.
[5] Open Group, "DCE 1.1 RPC specification," http://www.opengroup.org/onlinepubs/009629399/, 1997, accessed Apr. 2007.
[6] F. Reverbel and M. Fleury, "The JBoss extensible server," in *Proc. ACM Middleware Conference*, 2003.
[7] Sun Microsystems, "Java 2 enterprise edition," http://java.sun.com/j2ee/, accessed Apr. 2007.
[8] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.
[9] A. Stevens *et al.*, "Xdoclet," http://xdoclet.sourceforge.net/, accessed Apr. 2007.
[10] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java application partitioning," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, LNCS 2374, 2002, pp. 178–204.
[11] ——, "Portable and efficient distributed threads for Java," in *ACM Middleware Conference*. Springer-Verlag, Oct. 2004, pp. 478–492.
[12] D. Weyns, E. Truyen, and P. Verbaeten, "Distributed threads in Java," in *Proc. International Symposium on Distributed and Parallel Computing (ISDPC)*, 2002.
[13] Sun Microsystems, "Remote method invocation specification," http://java.sun.com/products/jdk/rmi/, 1997, accessed Apr. 2007.
[14] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad, "Efficient implementations of Java remote method invocation (RMI)," in *Proc. of Usenix Conference on Object-Oriented Technologies and Systems (COOTS98)*, 1998.
[15] M. Philippsen, B. Haumacher, and C. Nester, "More efficient serialization and RMI for Java," *Concurrency: Practice and Experience*, vol. 12, no. 7, pp. 495–518, May 2000.
[16] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat, "An efficient implementation of Java's remote method invocation," in *Proc. of ACM Symposium on Principles and Practice of Parallel Programming*, May 1999.
[17] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient Java RMI for parallel programming," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 6, pp. 747–775, Nov. 2001.
[18] G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski, "Reflective remote method invocation," *Concurrency: Practice and Experience*, vol. 10, no. 11-13, pp. 911–926, Sep.-Nov. 1998.
[19] Object Management Group, "Objects by value specification," http://www.omg.org/cgi-bin/doc?orbos/98-01-18.pdf, Jan. 1998, accessed Apr. 2007.
[20] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," in *Proc. 13th ACM Symposium on Operating Systems Principles*, Oct. 1991, pp. 152–164.
[21] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, and M. F. Kaashoek, "Performance evaluation of the Orca shared-object system," *ACM Trans. on Computer Systems*, vol. 16, no. 1, pp. 1–40, Feb. 1998.
[22] Y. Aridor, M. Factor, and A. Teperman, "cJVM: a single system image of a JVM on a cluster," in *Proc. International Conference on Parallel Programming (ICPP)*, 1999.
[23] M. Surdeanu and D. I. Moldovan, "Design and performance of a distributed Java virtual machine," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 6, pp. 611–627, Jun. 2002.
[24] W. Yu and A. Cox, "Java/DSM: A platform for heterogeneous computing," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1213–1224, 1997.
[25] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano, "A bytecode translator for distributed execution of legacy Java software," in *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Jun. 2001.
[26] A. Spiegel, "Automatic distribution of object-oriented programs," Ph.D. dissertation, FU Berlin, FB Mathematik und Informatik, Dec. 2002.
[27] B. Haumacher, J. Reuter, and M. Philippsen, "JavaParty: A distributed companion to java," http://wwwipd.ira.uka.de/JavaParty/, accessed Apr. 2007.
[28] M. Philippsen and M. Zenger, "JavaParty—transparent remote objects in Java," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1125–1242, 1997.
[29] M. Dahm, "Doorastha—a step towards distribution transparency," in *Proc. Java Informations Tage (JIT)/Net.ObjectDays 2000*, 2000.
[30] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, "A note on distributed computing," Technical Report, Sun Microsystems Laboratories, SMLI TR-94-29, Nov. 1994.

Eli Tilevich is an Assistant Professor in the Computer Science Department at Virginia Tech. He earned his B.A. degree from Pace University, M.S. from New York University, and Ph.D. from Georgia Tech. He is a member of the IEEE, and his research interests are in the systems and languages end of software engineering, spanning software technology, object-oriented programming, and distributed systems.

Yannis Smaragdakis is an Associate Professor of Computer Science at the University of Oregon. He earned his B.Sc. degree from the University of Crete and his Ph.D. from the University of Texas at Austin. He is a senior member of the IEEE and his interests are in the programming languages and systems side of software engineering.