DeXteR – An Extensible Framework for Declarative Parameter Passing in Distributed Object Systems

Sriram Gopal, Wesley Tansey, Gokulnath C. Kannan, and Eli Tilevich

Department of Computer Science
Virginia Tech, Blacksburg, VA 24061, USA
{sriramg,tansey,gomaths,tilevich}@cs.vt.edu

Abstract. In modern distributed object systems, reference parameters are passed to a remote method based on their runtime type. We argue that such type-based parameter passing is limiting with respect to expressiveness, readability, and maintainability, and that parameter passing semantics should be decoupled from parameter types. We present *declarative parameter passing*, an approach that fully decouples parameter passing semantics from parameter types in distributed object systems. In addition, we describe DeXteR, an extensible framework for transforming a type-based remote parameter passing model to a declaration-based model transparently. Our framework leverages aspect-oriented and generative programming techniques to enable adding new remote parameter passing semantics, without requiring detailed understanding of the underlying middleware implementation. Our approach is applicable to both application and library code and incurs negligible performance overhead. We validate the expressive power of our framework by adding several non-trivial remote parameter passing semantics (i.e., copy-restore, lazy, streaming) to Java RMI.

Keywords: Extensible Middleware, Metadata, Parameter Passing, Aspect Oriented Programming, Declarative Programming.

1 Introduction

Organizations have hundreds of workstations connected into local area networks (LANs) that stay unused for hours at a time. Consider leveraging these idle computing resources for distributed scientific computation. Specifically, we would like to set up an ad-hoc grid that will use the idle workstations to solve bioinformatics problems. The ad-hoc grid will coordinate the constituent workstations to align, mutate, and cross DNA sequences, thereby solving a computationally intensive problem in parallel.

Each workstation has a standard Java Virtual Machine (JVM) installed, and the LAN environment makes Java RMI a viable distribution middleware choice. As a distributed object model for Java, RMI simplifies distributed programming by exposing remote method invocations through a convenient programming

V. Issarny and R. Schantz (Eds.): Middleware 2008, LNCS 5346, pp. 144–163, 2008.
© IFIP International Federation for Information Processing 2008

model. In addition, the synchronous communication model of Java RMI aligns well with the reliable networking environment of a LAN.

The bioinformatics application follows a simple Master-Worker architecture, with classes `Sequence`, `SequenceDB`, and `Worker` representing a DNA sequence, a collection of sequences, and a worker process, respectively. Class `Worker` implements three computatinally-intensive methods: `align`, `cross`, and `mutate`.

```
interface WorkerInterface {
  void align(SequenceDB allSeqs, SequenceDB candidates, Sequence toMatch);
  Sequence cross(Sequence s1, Sequence s2);
  void mutate(SequenceDB seqs);
}

class Worker implements WorkerInterface { ... }
```

The `align` method iterates over a collection of candidate sequences (`candidates`), adding to the global collection (`allSeqs`) those sequences that satisfy a minimum alignment threshold. The `cross` method simulates the crossing over of two sequences (e.g., during mating) and returns the offspring sequence. Finally, the `mutate` method simulates the effect of a gene therapy treatment on a collection of sequences, thereby mutating the contents of every sequence in the collection.

Consider using Java RMI to distribute this application on an ad-hoc grid, so that multiple workers could solve the problem in parallel. To ensure good performance, we need to select the most appropriate semantics for passing parameters to remote methods. However, as we argue next, despite its Java-like programming model, RMI uses a different remote parameter passing model that is *type-based*. That is, the runtime type of a reference parameter determines the semantics by which RMI passes it to remote methods. We argue that this parameter passing model has serious shortcomings, with negative consequences for the development, understanding, and maintenance of distributed applications.

Method `align` takes two parameters of type `SequenceDB`: `allseqs` and `candidates`. `allseqs` is an extremely large global collection that is being updated by multiple workers. We, therefore, need to pass it by *remote-reference*. `candidates`, on the other hand, is a much smaller collection that is being used only by a single worker. We, therefore, need to pass it by *copy*, so that its contents can be examined and compared efficiently. To pass parameters by *remote-reference* and by *copy*, the RMI programmer has to create subclasses implementing marker interfaces `Remote` and `Serializable`, respectively. As a consequence, method `align`'s signature must be changed as well. Passing `allSeqs` by *remote-reference* requires the type of `allSeqs` to become a remote interface. Finally, examining the declaration of the remote method `align` would give no indication about how its parameters are passed, forcing the programmer to examine the declaration of each parameter's type. In addition, in the absence of detailed source code comments, the programmer has no choice but to examine the logic of the entire slice [3] of a distributed application that can affect the runtime type of a remote parameter.

Method `mutate` mutates the contents of every sequence in its `seqs` parameter. Since the client needs to use the mutated sequences, the changes have to be reflected in the client's JVM. The situation at hand renders passing by *remote-reference* ineffective, since the large number of remote callbacks is likely to incur a significant performance overhead. One approach is to pass `seqs` by *copy-restore*, a semantics which efficiently approximates *remote-reference* under certain assumptions [22].

Because Java RMI does not natively support *copy-restore*, one could use a custom implementation provided either by a third-party vendor or an in-house expert programmer. Mainstream middleware, however, does not provide programming facilities for such extensions. Thus, adding a new semantics would not only require a detailed understanding of the RMI implementation, but also sufficient privileges to modify the Java installation on each available idle workstation.

Finally, consider the task of maintaining the resulting ad-hoc grid distributed application. Assume that `SequenceDB` is a remote type in one version of the application, such that RMI will pass all instances `SequenceDB` by *remote-reference.* However, if a maintenance task necessitates passing some instance of `SequenceDB` using different semantics, the `SequenceDB` type would have to be changed. Nevertheless, if `SequenceDB` is part of a third-party library, it may not be subject to modification by the maintenance programmer.

To overcome the limitations of a type-based remote parameter passing model, we present an alternative, *declarative* model. We argue that remote parameter passing should resemble that of local parameter passing in mainstream programming languages. Following this paradigm, a passing mechanism for each parameter is specified at the declaration of each remote method. By decoupling parameter passing from parameter types, our approach increases expressiveness, improves readability, and eases maintainability.

Unsurprisingly, mainstream programming languages such as C, C++, and C# express the choice of parameter passing mechanisms through method declarations with special tokens instead of types. For example, by default objects in C++ are passed by *value*, but inserting the `&` token after the type of a parameter signals the by *reference* mechanism. We argue that distributed object systems should adhere to a similar declarative paradigm for remote method calls, but properly designed for distributed communication.

While Java always uses the by *value* semantics for local calls, we argue that distributed communication requires a richer set of semantics to ensure good performance and to increase flexibility. We also argue that IDL-based distributed object systems such as CORBA [11] and DCOM [1] with their `in`, `out`, and `inout` parameter modes stop short of a fully declarative parameter model and are not extensible.

Recognizing that many existing distributed applications are built upon a type-based model, we present a technique for transforming a type-based remote parameter passing model to use a declaration-based one. Our technique transforms parameter passing functionality transparently, without any changes to the

underlying distributed object system implementation, ensuring cross-platform compatibility and ease of adoption. With Java RMI as our example domain, we combine aspect-oriented and generative techniques to retrofit its parameter passing functionality. Our approach is equally applicable to application classes, system classes, and third-party libraries.

In addition, we argue that a declarative model to remote parameter passing simplifies adding new semantics to an existing distributed object model. Specifically, we present an extensible plug-in-based framework, through which third-party vendors or in-house expert programmers can seamlessly extend a native set of remote parameter passing semantics with additional semantics. Our framework allows such extension in the application space, without modifying the JVM or its runtime classes. As a validation, we used our framework to extend the set of available parameter passing semantics of RMI with several non-trivial state-of-the-art semantics, introduced earlier in the literature both by us [22] and others [4,7,25].

One of the new semantics we implemented using our framework is an optimization of our own algorithm for *copy-restore* [22]. In the original implementation, the server sends back a complete copy of the parameter to the restore stage of the algorithm on the client, which is inefficient in high-latency, low-bandwidth networking environments. The implemented optimized version of the *copy-restore* algorithm, which we call *copy-restore with delta*, efficiently identifies and encodes the changes made by the server to the parameter, sending to the client only the resulting delta. Because the original *copy-restore* algorithm performs better in high-bandwidth networks, our extensible framework makes it possible to use different versions of the *copy-restore* algorithm for different remote calls in the same application.

We believe that the technical material presented in this paper makes the following novel contributions:

- A clear exposition of the shortcomings of type-based parameter passing models in modern distributed object systems such as CORBA, Java RMI, and .NET Remoting.
- An alternative declarative parameter passing approach that offers multiple design and implementation advantages.
- An extensible framework for retrofitting standard RMI applications to take advantage of our declaration based model and for extending the RMI native set of parameter passing semantics.
- An enhanced *copy-restore* mode of remote parameter passing, offering performance advantages for low bandwidth, high latency networks.

The rest of this paper is structured as follows. Section 2 presents DeXteR, our extensible framework. Section 3 describes how we used DeXteR to add several non-trivial parameter passing semantics to RMI. Section 4 discusses the advantages and constraints of our approach. Section 5 discusses related work. Finally, Section 6 outlines future work directions and conclusions.

2 The *DeXter* Framework

This section discusses the design and implementation of **DeXteR** (**D**eclarative **Ext**ensible **R**emote Parameter Passing), a framework for declarative remote parameter passing.

2.1 Framework Overview

DeXteR implements declaration-based parameter passing semantics on top of standard Java RMI, without modifying its implementation. DeXteR uses a plug-in based architecture and treats remote parameter passing as a distributed cross-cutting concern. Each parameter passing style is an independent plugin component.

DeXteR uses the Interceptor Pattern [18] to expose the invocation context explicitly on the client and the server sites. While Interceptors have been used in several prior systems [8] to introduce orthogonal cross-cutting concerns such as logging and security, the novelty of our approach lies in employing Interceptors to transform and enhance the core functionality of a distributed object system, its remote parameter passing semantics.

Figure 1 depicts the overall translation strategy employed by DeXteR. The rank-and-file (i.e., application) programmer annotates an RMI application with the desired remote parameter passing semantics. The annotations processor takes the application source code as input, and extracts the programmer's intent. The extracted information parameterizes the source code generator, which encompasses the framework-specific code generator and the extension-specific code

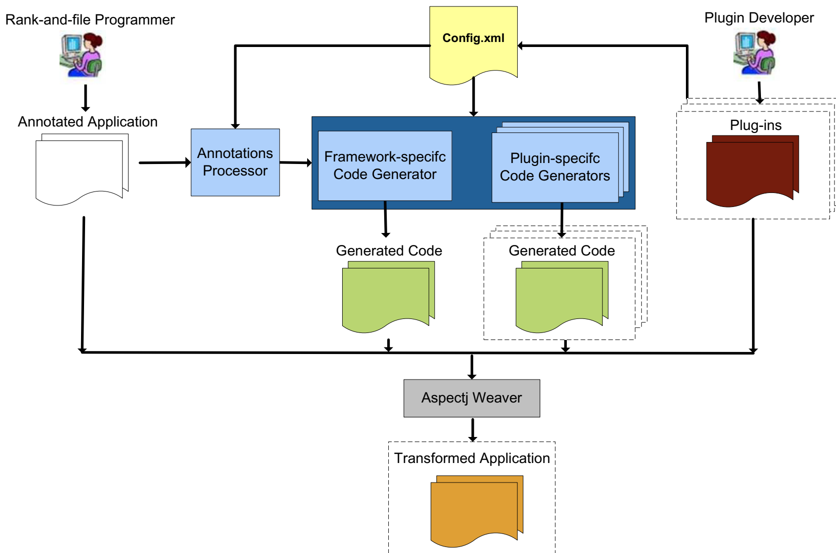

Fig. 1. Development and Deployment Process

generators. The framework-specific code generator synthesizes the code for the client and the server interceptors using aspects. The extension-specific code generators synthesize the code pertaining to the translation strategy for supporting a specific parameter passing semantics. DeXteR compiles the generated code into bytecode, and the resulting application uses standard Java RMI, only with a small AspectJ runtime library as an extra dependency. The generated aspects are weaved into the respective classes at load-time, thereby redirecting the invocation to the framework interceptors at both the local and the remote sites.

2.2 Framework API

DeXteR provides interception points for parameter passing plugins in the form of the `InterceptionPoint` interface. Developing a new plugin involves implementing this interface and identifying the interception points of interest, providing the functionality at these interception points, and registering the plugin with the framework.

```
interface InterceptionPoint {
   // Interception points on client-side
   Object [] argsBeforeClientCall(Object target, Object [] args);
   Object [] customArgsBeforeClientCall(Object target);
   Object retAfterClientCall(Object target, Object ret);
   void customRetAfterClientCall(Object target, Object [] customRets);

   // Interception points on server-side
   Object [] argsBeforeServerCall(Object target, Object [] args);
   void customArgsBeforeServerCall(Object target, Object [] customArgs);
   Object retAfterServerCall(Object target, Object ret);
   Object[] customRetAfterServerCall(Object target);

   // Plugin-specific code generator
   void generate(AnnotationInfo info);
}
```

The above interface exposes the invocation context of a remote call at different points of its control-flow on both the client and server sites. DeXteR exposes to a plugin only the invocation context pertaining to the corresponding parameter passing annotation. For example, plugin X obtains access only to those remote parameters annotated with annotation `X`. DeXteR enables plugins to modify the original invocation arguments as well as to send custom information between the client- and the server-side extensions. The custom information is simply piggy-backed to the original invocation context.

2.3 Implementation Details

The interception is implemented by combining aspect-oriented and generative programming techniques. Specifically, DeXteR uses AspectJ to add extra methods to RMI remote interface, stub, and server implementation classes for each

remote method. These methods follow the Proxy pattern to interpose the logic required to support various remote parameter passing strategies. Specifically, the flow of a remote call is intercepted to invoke the plugins with the annotated parameters, and the modified set of parameters is obtained. The intercepted invocation on the client site is then redirected to the added extra method on the server. The added server method reverses the process, invoking the parameter passing style plugins with the modified set of parameters provided by their client-side peers. The resulting parameters are used to make the invocation on the actual server method. A similar process occurs when the call returns, in order to support different passing styles for return types.

For each remote method, DeXteR injects a wrapper method into the remote interface and the server implementation using *inter-type declarations*, and pointcuts on the execution of that method in the stub (i.e., implemented as a dynamic proxy) to provide a wrapper in the form of an *around* advice. All the AspectJ code that provides the interception functionality is automatically generated at compile time, based on the remote method's signature.

2.4 Bioinformatics Example Revisited

DeXteR enables the programmer to express remote parameter passing semantics exclusively by annotating remote method declarations with the intended passing semantics. A distributed version of the bioinformatics application from section 1 can be expressed using DeXteR as follows. The different parameter passing semantics are introduced without affecting the semantics of the centralized version of the application.

```
public interface WorkerInterface extends Remote
{
   void align(@RemoteRef SequenceDB matchingSeqs,
              @Copy SequenceDB candidates,
              @Copy Sequence toMatch) throws RemoteException;

   @Copy Sequence cross(@Copy Sequence s1, @Copy Sequence s2)
                        throws RemoteException;
   void mutate(@CopyRestore SequenceDB seqs)
                        throws RemoteException;
}
```

Since remote parameter passing annotations are part of a remote method's signature, they must appear in both the method declaration in the remote interface and the method definitions in all remote classes implementing the interface. This requirement ensures that the client is informed about how remote parameters will be passed, and it also allows for safe polymorphism (i.e., the same remote interface may have multiple remote classes implementing it). We argue, however, that this requirement should not impose any additional burden on the programmer. A modern IDE such as Eclipse, NetBeans, or Visual Studio should be able to reproduce the annotations when providing method stub implementations for remote interfaces.

3 Supporting Parameter Passing Semantics

This section describes the strategies for implementing several non-trivial parameter passing semantics previously proposed in the research literature [22,7,25,4] as DeXteR plugins. We restrict our description to parameters, as the strategies for handling return types are identical.

To demonstrate the power and expressiveness of our approach, we chose the semantics that have very different implementation requirements. While the lazy semantics requires flexible proxying on-demand, copy-restore requires passing extra information between the client and the server. Despite the contrasting nature of these semantics, we were able to encapsulate all their implementation logic inside their respective plugins and easily deploy them using DeXteR.

3.1 Lazy Semantics

Lazy parameter passing [7], also known as *lazy pass-by-value*, provides a useful semantics for asynchronous distributed environments, specifically in P2P applications. It works by passing the object initially by reference and then transferring it by value either upon first use (*implicitly lazy*) or at a point dictated by the application (*explicitly lazy*). More precisely, lazy parameter passing defines *if and when exactly an object is to be passed by value.*

The translation strategy for passing reference objects by *lazy* semantics involves using the plugin-specific code generator. As our aim is to decouple parameter types from the semantics by which they are passed, to pass a parameter of

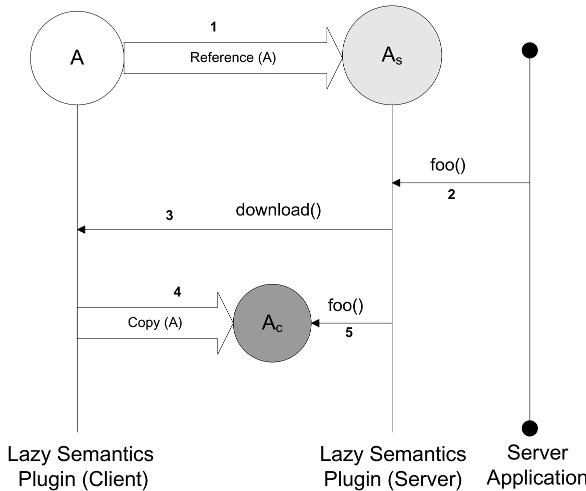

Fig. 2. Lazy Semantics Plugin Interaction Diagram (A: Serializable Object; A_s: Stub of A; A_c: Copy of A; (1) A is passed from client to server; (2) Server invokes `foo()` on stub A_s; (3) Server plugin calls `download()` on client plugin; (4) Client plugin sends a copy of A, A_c; (5) Server plugin calls `foo()` on A_c.)

type A by *lazy* semantics does not require defining any special interface nor A implementing one. Instead, the plugin-specific code generator generates a Remote interface, declaring all the accessible methods of A. To make our approach applicable for passing both application and system classes, we deliberately avoid making any changes to the bytecode of a parameter's class A. Instead, we use a delegating dynamic proxy (e.g., A_DynamicProxy) for the generated Remote interface (e.g., AIface) and generate a corresponding server-side proxy (e.g., A_ServerProxy) that is type-compatible with the parameter's class A. As is common with proxy replacements for remote communication [6], all the direct field accesses of the *remote-reference* parameter on the server are replaced with accessor and mutator methods.[1]

In order to enable obtaining a copy of the remote parameter (at some point in execution), the plugin inserts an additional method download() in the generated remote interface AIface, the client proxy A_DynamicProxy and the server proxy A_ServerProxy.

```
class A {
  public void foo() {...}
}

// Generated remote interface
interface AIface extends Remote {
   public void foo() throws RemoteException;
   public A download() throws RemoteException;
}

// Generated client proxy
class A_DynamicProxy implements AIface {
   private A remoteParameter;

   public A download() {
      // serialize remoteParameter
   }

   public void foo() throws RemoteException { ... }
}

// Generated server proxy
class A_ServerProxy extends A {
   private A a;
   private AIface stub;

   public A_ServerProxy(AIface stub) {
      this.stub = stub;
   }
```

[1] Replacing direct fields accesses with methods has become such a common transformation that AspectJ [13] provides special fields access pointcuts (i.e., set, get) to support it.

```
    synchronized void download() {
        // Obtain a copy of the remote parameter
        a = stub.download();
    }

    public void foo() {
        // Dereference the stub
        stub.download();
        // Invoke the method on the copy
        a.foo();
    }
}
```

Any invocation made on the parameter (i.e., server proxy) by the server results in a call to its `synchronized` `download()` method, if a local copy of the parameter is not yet available. The `download()` method of the server proxy relays the call to the `download()` method of the enclosed client proxy with the aim of obtaining a copy of the remote parameter.

The client proxy needs to serialize a copy of the parameter. However, passing a remote object (i.e., one that implements a `Remote` interface) by *copy* presents a unique challenge, as type-based parameter passing mechanisms are deeply entangled with Java RMI. The RMI runtime replaces the object with its stub, effectively forcing pass by *remote-reference*. The plugin-generated code overrides this default functionality of Java RMI by rendering a given remote object as a memory buffer using Serialization. This technique effectively "hides" the remote object, as the RMI runtime transfers memory buffers without inspecting or modifying their content. The "hidden" remote object can then be extracted from the buffer on the server-side and used as a parameter. Once the copy is obtained, all subsequent invocations made on the parameter (i.e., server proxy) are delegated to the local copy of the parameter.

Thus, passing an object of type `A` as a parameter to a remote method will result in the client-side plugin replacing it with its type-incompatible stub. The server-side plugin wraps this type-incompatible stub into the generated server-side proxy that is type-compatible with the original remote object.

We note that a subset of the strategies described above is used for supporting the native semantics *copy* and *remote-reference*.

3.2 Copy Restore Semantics

A semantics with a different set of implementation requirements than that of *lazy* parameter passing is the *copy-restore* semantics. It copies a parameter to the server and then restores the changes to the original object in place (i.e., preserving client-side aliases).

Implementing the *copy-restore* semantics involves tracing the invocation arguments and restoring the changes made by the server after the call. The task

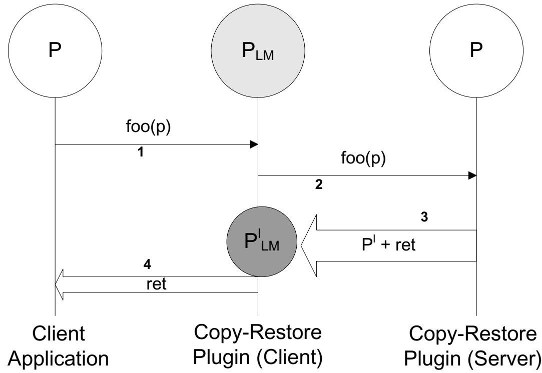

Fig. 3. Copy-Restore Semantics Plugin Interaction Diagram (P: Set of parameters passed to `foo`; P_{LM}: Linear map of parameters; P^l: Modified parameters (restorable data); *ret*: values returned by the invocation; $P^1{}_{LM}$: Modified linear map; (1) The client invokes method `foo()` passing parameter p; (2) The client-side plugin constructs a linear map P_{LM} and calls the original `foo(p)`; (3) Server-side plugin invokes `foo` and returns modified parameters P^l and the return value *ret*; (4) Changes restored and the return value *ret* is passed to the client.)

is simplified by the well-defined hook points provided by the framework. Prior to the remote method invocation, the *copy-restore* plugin obtains a copy of the parameter `A` and does some pre-processing on both the client and the server sites. The invocation then resumes and the server mutates the parameter during the call. Once the call completes, the server-side plugin needs to send back the changes to the parameter made by the server to its client-side peer. This is accomplished using the custom information passing facility provided by the framework. The client-side plugin uses this information from its server-side peer to restore the changes to the parameter A in the client's JVM.

3.3 Copy Restore with Delta Semantics

For single-threaded clients and stateless servers, *copy-restore* makes remote calls indistinguishable from local calls as far as parameter passing is concerned [22]. However, in a low bandwidth high latency networking environment, such as in a typical wireless network, the reference *copy-restore* implementation may be inefficient. The potential inefficiency lies in the restore step of the algorithm, which always sends back to the client an entire object graph of the parameter, no matter how much of it has been modified by the server. To optimize the implementation of *copy-restore* for low bandwidth, high latency networks, the restore step can send back a "delta" structure by encoding the differences between the original and the modified objects. The necessity for such an optimized *copy-restore* implementation again presents a compelling case for extensibility and flexibility in remote parameter passing.

The following pseudo-code describes our optimized *copy-restore* algorithm, which we term *copy restore with delta*:

1. Create and keep a linear map of all the objects transitively reachable from the parameter.
2. On the server, again create a linear map, `Lmap1`, of all the objects transitively reachable from the parameter.
3. Deep copy `Lmap1` to an isomorphic linear map `Lmap2`.
4. Execute the remote method, modifying the parameter and `Lmap1`, but not `Lmap2`.
5. Return `Lmap1` back to the client; when serializing `Lmap1`, encode the changes to the parameter by comparing with `Lmap2` as follows:
   (a) Write as is each changed existing object or a newly added object.
   (b) Write its numeric index in `Lmap1` for each unchanged existing object.
6. On the client, replay the encoded changes, using the client-side linear map to retrieve the original old objects at the specified indexes.

Creating Linear Map. A linear map of objects transitively reachable from a reference argument is obtained by tapping into serialization, recording each encountered object during the traversal. In order not to interfere with garbage collection, all linear maps use weak references.

Calculating Delta. The algorithm encodes the delta information efficiently using a handle structure shown below.

```
class Handle{
   int id;
   ArrayList<Long> chId;
   ArrayList<Long> chScript;
   ArrayList<Object> chObject;
}
```

The identifier `id` refers to the position of an object in the client site linear map. The change indicator `chId` identifies the modified member fields using a bit level encoding. `chScript` contains the changes to be replayed on the old object. For a primitive field, its index simply contains the new value, whereas for an object field, its index points to `chObject`, which contains the modified references.

Restoring Changes. For each de-serialized handle on the client, the corresponding old object is obtained from the client's linear map using the handle identifier `id`. The handle is replaced with the old object, and the changes encoded in the handle are replayed on it. Following the change restoration, garbage collection reclaims the unused references.

As a concrete example of our algorithm, consider a simple binary tree, `t`, of integers. Every node in the tree has three fields: `data`, `left`, and `right`. A subset of the tree is aliased by non-tree pointers `alias1` and `alias2`. Consider a remote method such as the one show below, to which tree `t` is passed as a parameter.

```
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 4 shows the sequence of steps involved in passing tree `t` by *copy restore with delta* and restoring the changes made by the remote method `alterTree` to the original tree.

We measured the performance gains of our algorithm over the original copy-restore by conducting a series of micro-benchmarks, varying the size of a binary

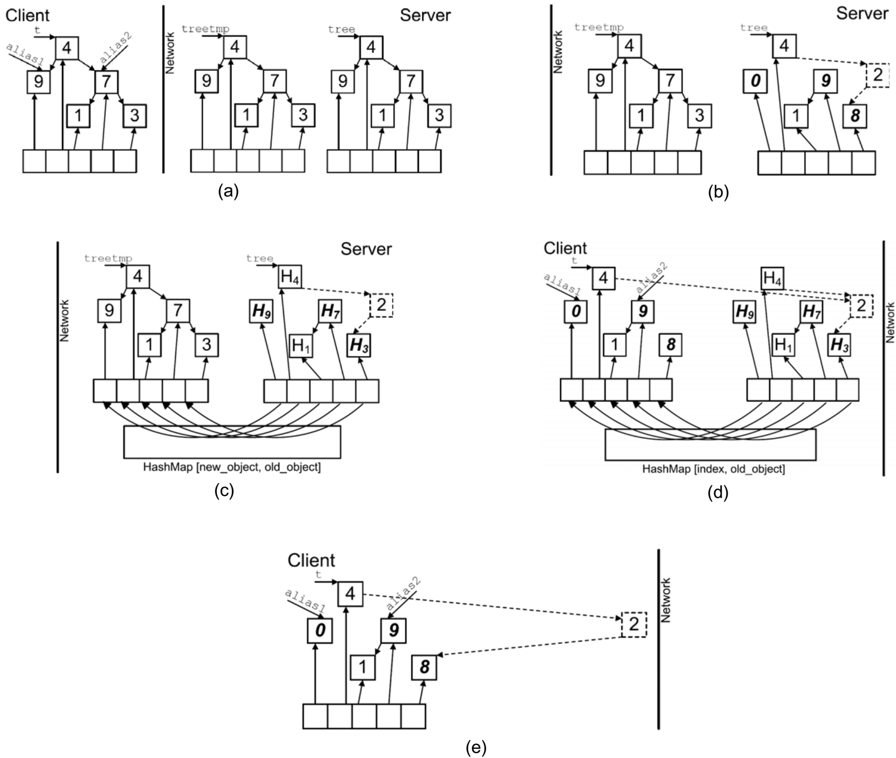

Fig. 4. *Copy-restore with delta* algorithm by example (a) State after step 3. (b) State after step 4. The remote procedure modified the parameter. (c) State during step 5. Copy the modified objects (even those no longer reachable through `tree`) back to the client; compute the delta script for modified objects using a hash map. (d) State during step 6. Replace the handles with the original old objects; replay the delta script to reflect changes. (e) State of the client side object after step 6.

tree and the amount of changes performed by the server. The benchmarks were run on Pentium 2.GHz (dual core) machines with 2 GB RAM, running Sun JVM version 1.6.0 on an 802.11b wireless LAN. Figure 5 shows the percentage of performance gain of copy-restore with delta over copy-restore. Overall, our experiments indicate that the performance gain is directly proportional to the size of the object graph and is inversely proportional to the amount of changes made to the object graph by the server.

By providing flexibility in parameter passing, DeXteR enables programmers to use different semantics or different variations of the same semantics as determined by the nature of the application. For instance, within the same application one can use regular *copy-restore* for passing small parameters and *copy-restore with delta* for passing large parameters.

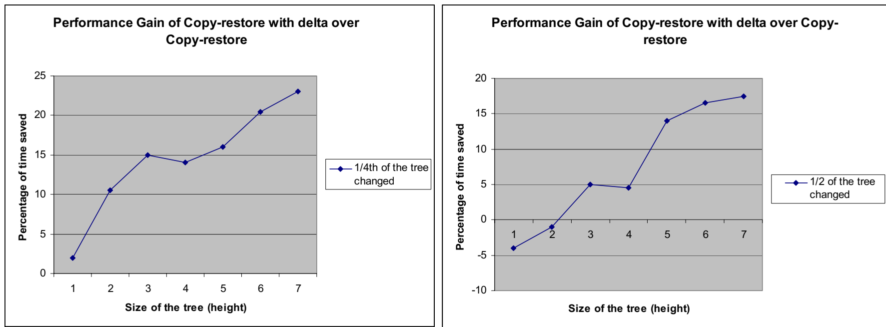

Fig. 5. Performance gain of copy-restore with delta over copy-restore

3.4 Other Semantics

Additional semantics we implemented using DeXteR include *streaming* [25], *parameter substitution a.k.a caching* [4], and some others. Due to space constraints, we do not explain them in detail.

DeXteR offers the advantages of supporting a wide variety of remote parameter passing semantics through a uniform API. Developments in hardware and software designs are likely to cause the creation of new parameter passing semantics. These semantics will leverage the new designs, but may be too experimental to be included in the implementation of a standard middleware system. DeXteR will allow the integration and use of these novel semantics at the application layer, without changing the underlying middleware. As a particular example, consider the introduction of massive parallelism into mainstream processors. Multiple cores will require the use of explicit parallelism to improve performance. Some facets of parameter passing are computation-intensive and can benefit from parallel processing. One can imagine, for instance, how marshaling could be performed in parallel, in which parts an object graph are serialized/deserialized by different cores.

4 Discussion

This section discusses the advantages of DeXteR as well as some of the constraints of our design.

4.1 Design Advantages

Expressing remote parameter passing choices as a part of the method declaration has several advantages over a type-based system. Specifically, a declarative approach increases expressiveness, improves readability, and eases maintainability. To further illustrate the advantages of our declarative framework, we compare and contrast our approach with that of Java RMI.

Expressiveness. Java RMI restricts expressiveness by assuming that all instances of the same type will be passed identically. Passing the same type using different semantics therefore requires creating subclasses implementing different marker interfaces and changing the method signature. By contrast, our approach does not require any new subclasses to be created or any changes to be made to the original method signature. Furthermore, under Java RMI, the programmer of the class has no simple way to enforce how the parameters are actually passed to its remote methods. The simple declarative style of our annotations makes enforcement of the parameter passing policies straightforward.

Readability. Examining the declaration of a remote method does not reveal any details about how its parameters are passed, forcing the programmer to examine each parameter type individually, which reduces readability and hinders program understanding. By contrast, our approach provides a single point of reference that explicitly informs the programmer about how remote parameters are passed.

Maintainability. An existing class may have to be modified to implement an interface before its instances can be passed as parameters to a remote method. This complicates maintainability as, in the case of third-party libraries, source code may be difficult or even impossible to modify. By contrast, our approach enables the maintenance programmer modify the semantics by simply specifying a different parameter passing annotation.

Extensibility. Even if the *copy-restore* semantics gets the attention of the Java community and is natively supported in the next version of Java, including new optimization mechanisms such as using *copy-restore with delta* would still mean modifying the underlying Java RMI implementation of both the client and the server. By contrast, our approach supports extending the native remote parameter passing semantics at the application-level, requiring absolutely no modifications to the underlying middleware.

Reusability. DeXteR also enables providing the parameter passing semantics as plugin libraries. Application programmers thus can obtain third-party plugins and automatically enhance their own RMI applications with the new parameter passing semantics.

Efficiency. Another advantage of our approach is its efficiency. That is, all the transformations described in Section 3 do not result in any additional overhead in using objects of type `A` until they are passed using a particular mode in an RMI call. This requires that one know exactly when an object of type `A` is used in this capacity. The insight that makes it possible to efficiently detect such cases is that the program execution flow must enter an RMI stub (dynamic proxy) for a remote call to occur.

To measure the overhead of DeXteR, we ran a series of microbenchmarks comparing the execution times of the DeXteR-based parameter passing semantics' implementations and their native counterparts, of which pass by *remote-reference* is of particular interest. In lieu of support for type-compatible dynamic proxies for classes in Java, our *remote-reference* DeXteR plugin emulates this functionality using a type-incompatible client-side dynamic proxy and a type-compatible server-side wrapper proxy. Thus, this emulated functionality introduces two new levels of indirection compared to the standard Java RMI implementation of this semantics. As any new level of indirection inherently introduces some performance overhead, it is important to verify that it is not prohibitively expensive.

To distill the pure overhead, we ran the benchmarks on a single machine. In the presence of network communication and added latency, the overhead incurred by the additional levels of local indirection would be dominated. Therefore, the results do not unfairly benefit our approach. The resulting overhead never exceeds a few percentage points of the total latency of a remote call executed on a single machine. Due to space constraints, we do not present the detailed results of this experiment here, but the interested reader can find them in reference [9]. In general, as the latency of a remote call is orders of magnitude greater than that of a local call, the overhead incurred by a DeXteR plugin adding a few simple local calls to a remote call should be negligible.

4.2 Design Constraints

Achieving the afore-mentioned advantages without changing the Java language required constraining our design in the following ways.

First, array objects are always passed by *copy* though the array elements could be passed using any desired semantics. While this is a limitation of our system, it is still nonetheless a strict improvement over standard RMI, which also passes array objects by *copy*, but passes array elements based on their runtime type.

Second, passing `final` classes (not extending `UnicastRemoteObject`) by *remote-reference* would entail either removing their `final` specifier or performing a sophisticated global replacement with an isomorphic type [23]. This requirement stems from our translation strategy's need to create a proxy subclass for *remote-reference* parameters, an impossibility for `final` classes. Since heavy transformations would clash with our design goal of simplicity, our approach issues a compile-time error to an attempt to pass an instance of a `final` class by *remote-reference*. Again, this limitation is also shared by standard RMI.

Finally, since our approach does not modify standard Java, it is not possible to support direct member field access for instances of system classes passed by

Table 1. Analysis of Java 6 JDK's public member fields (some overlap exists due to `Exception` classes spanning multiple packages)

Classes Analyzed	Total	Classes With Public Fields	Total Public Fields
All User-Accessible Classes	2732	57	123
GUI Classes	913	15	65
Exception Classes	364	33	34
RMI Classes	58	22	22
Java Bean Classes	56	3	3

remote-reference. While this is a conceptual problem, an analysis of the Java 6 library shown in Table 1 indicates that this is not a practical problem. For our purposes, we analyzed the `java.*` and `javax.*` classes, as they are typically the ones mostly used by application developers. As the table demonstrates, approximately 1% of classes contain non-final member fields. However, the vast majority of these classes are either GUI or sound components, SQL driver descriptors, RMI internal classes, or exception classes, and as such, are unlikely to be passed by *remote-reference*. Additionally, the classes in `java.beans.*` provide getter methods for their public fields, thereby not requiring direct access. The conclusion of our analysis is that only one (`java.io.StreamTokenizer`) of the more than 5,500 analyzed classes could potentially pose a problem, with two public member fields not accessible by getter methods.

5 Related Work

The body of research literature on distributed object systems and separation of concerns is extremely large and diverse. The following discusses only closely-related state of the art.

Separation of Concerns. Several language-based and middleware-based approaches address the challenges in modeling cross-cutting concerns.

Proxies and Wrappers [20] introduce late bound cross-cutting features, though in an application-specific manner.

Aspect Oriented Programming (AOP) [14] is a methodology for modularizing cross-cutting concerns. Several prior AOP approaches aim at improving various properties of middleware systems, with the primary focus on modularization [5,26].

Java Aspect Components (JAC) [17] and DJCutter [15] support distributed AOP. JAC framework enables the dynamic adding or removing of an advice. DJCutter extends AspectJ with *remote pointcuts*, a special language construct for developing distributed systems. DeXteR could use these approaches as an alternative to AspectJ.

A closely related work is the DADO [24] system for programming cross-cutting features in distributed heterogeneous systems. Similar to DeXteR, DADO uses hook-based extension patterns. It employs a pair of user-defined adaplets,

explicitly modeled using IDL for expressing the cross-cutting behavior. To accommodate heterogeneity, DADO employs a custom DAIDL (an IDL extension) compiler, runtime software extensions, and tool support for dynamically retrofitting services into CORBA applications. DADO uses the Portable Interceptor approach for triggering the advice for cross-cutting concerns, which do not modify invocation arguments and return types. Thus, using DADO to change built-in remote parameter passing semantics would not eliminate the need for binary transformations and code generation.

Remote Parameter Passing. Multi-language distributed object systems such as CORBA [11], DCOM [1], etc., use an Interface Definition Language (IDL) to express how parameters are passed to remote methods. Each parameter in a remote method signature is associated with keywords `in`, `out`, and `inout` designating the different passing options. This approach however, does not completely decouple parameter passing from parameter types. When the IDL interface is mapped to a concrete language, the generated implementation still relies on a type-based parameter passing model of the target language. Specifically, in mapping IDL to Java [12], an IDL *valuetype* maps to a `Serializable` class, which is always passed by *copy*. Conversely, an IDL *interface* maps to a `Remote` class, which is always passed by *remote-reference*. Additionally, even if we constrain parameters to *valuetypes* only, the mapped implementation will generate different types based on the keyword modifiers present [10]. Thus, remote parameter passing in IDL-based distributed object systems is neither fully declarative, nor it is extensible.

.NET Remoting [16] for C# also follows a mixed approach to remote parameter passing. It supports the parameter-passing keywords *out* and *ref*. However, the *ref* keyword designates pass by *value-result* in remote calls rather than the standard pass by *reference* in local calls. This difference in passing semantics may lead to the introduction of subtle inconsistencies when adapting a centralized program for distributed execution. Furthermore, in the absence of any optional parameter passing keywords, a reference object is passed based on the parameter type. While this approach shares the limitations of Java RMI, *remote-reference* proxies are type-compatible stubs, which provide full access to the remote object's fields. Therefore, while the parameter passing model of .NET Remoting contains some declarative elements, it has shortcomings and is not extensible.

Doorastha [2] represents a closely related piece of work on increasing the expressiveness of distributed object systems. It aims at providing distribution transparency by enabling the programmer to annotate a centralized application with distribution tags such as *globalizable* and *by-refvalue*, and using a specialized compiler for processing the annotations to provide fine-grained control over the parameter passing functionality. While influenced by the design of Doorastha, our approach differs in the following ways. First, Doorastha does not completely decouple parameter passing from the parameter types, as it requires annotating classes of remote parameters with the desired passing style. Unannotated remote parameters are passed based on their type. Second, Doorastha does not support extending the default set of parameter passing modes. Finally, Doorastha

requires a specialized compiler for processing the annotations. While Doorastha demonstrates the feasibility of many of our approach's features, we believe our work is the first to present a comprehensive argument and design for a purely declarative and extensible approach to remote parameter passing.

6 Future Work and Conclusions

A promising future work direction is to develop a declaration-based distributed object system for an emerging object-oriented language, such as *Ruby* [21], utilizing its advanced language features such as built-in aspects, closures, and coroutines. Despite its exploratory nature and the presence of advanced features, Ruby's distributed object system, *DRuby* [19], does not significantly differ from Java RMI.

We presented a framework for declarative parameter passing in distributed object systems as a better alternative to type-based parameter passing. We described how a declarative parameter passing model with multiple different semantics can be efficiently implemented on top of a type-based parameter passing model using our extensible framework, DeXteR. We believe that our framework is a powerful distributed programming platform and an experimentation facility for research in distributed object systems.

Availability. DeXteR can be downloaded from `http://research.cs.vt.edu/vtspaces/dexter`.

Acknowledgments. The authors would like to thank Godmar Back, Doug Lea, Naren Ramakrishnan, and the anonymous reviewers, whose comments helped improve the paper. This research was supported by the Computer Science Department at Virginia Tech.

References

1. Brown, N., Kindel, C.: Distributed Component Object Model Protocol–DCOM/1.0 1998, Redmond, WA (1996)
2. Dahm, M.: Doorastha—a step towards distribution transparency. In: Proceedings of the Net. Object Days 2000 (2000)
3. De Lucia, A., Fasolino, A.R., Munro, M.: Understanding function behaviours through program slicing. In: 4th IEEE Workshop on Program Comprehension, pp. 9–18 (1996)
4. Eberhard, J., Tripathi, A.: Efficient Object Caching for Distributed Java RMI Applications. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 15–35. Springer, Heidelberg (2001)
5. Eichberg, M., Mezini, M.: Alice: Modularization of Middleware using Aspect-Oriented Programming. In: Gschwind, T., Mascolo, C. (eds.) SEM 2004. LNCS, vol. 3437. Springer, Heidelberg (2005)
6. Eugster, P.: Uniform proxies for Java. In: OOPSLA 2006: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 139–152. ACM Press, New York (2006)

7. Eugster, P.T.: Lazy Parameter Passing. Technical report, Ecole Polytechnique Fédérale de Lausanne, EPFL (2003)
8. Fleury, M., Reverbel, F.: The JBoss Extensible Server. In: International Middleware Conference (2003)
9. Gopal, S.: An extensible framework for annotation-based parameter passing in distributed object systems. Master's thesis, Virginia Tech. (June 2008)
10. Object Management Group. Objects by value. document orbos/98-01-18, Framingham, MA (1998)
11. Object Management Group. The common object request broker: Architecture and specification, Framingham, MA (1998)
12. Object Management Group. IDL to Java language mapping specification, Framingham, MA (2003)
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–355, 110. Springer, Heidelberg (2001)
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241. Springer, Heidelberg (1997)
15. Nishizawa, M., Chiba, S., Tatsubori, M.: Remote pointcut: a language construct for distributed AOP. In: Proceedings of the 3rd international conference on Aspect-oriented software development, pp. 7–15 (2004)
16. Obermeyer, P., Hawkins, J.: Microsoft .NET Remoting: A Technical Overview. MSDN Library (July 2001)
17. Pawlak, R., Seinturier, L., Duchien, L., Florin, G., Legond-Aubry, F., Martelli, L.: JAC: an aspect-based distributed dynamic framework. Software Practice and Experience 34(12), 1119–1148 (2004)
18. Schmidt, D.C., Rohnert, H., Stal, M., Schultz, D.: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. John Wiley & Sons, Inc., New York (2000)
19. Seki, M.: DRuby–A Distributed Object System for Ruby (2007), `http://www.ruby-doc.org/stdlib/libdoc/drb/`
20. Souder, T.S., Mancoridis, S.: A Tool for Securely Integrating Legacy Systems into a Distributed Environment. In: Working Conference on Reverse Engineering, pp. 47–55 (1999)
21. Thomas, D., Hunt, A.: Programming Ruby. Addison-Wesley, Reading (2001)
22. Tilevich, E., Smaragdakis, Y.: NRMI: Natural and Efficient Middleware. IEEE Transactions on Parallel and Distributed Systems, 174–187 (February 2008)
23. Tilevich, E., Smaragdakis, Y.: J-Orchestra: Automatic Java Application Partitioning. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374. Springer, Heidelberg (2002)
24. Wohlstadter, E., Jackson, S., Devanbu, P.: DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In: Proceedings of the International Conference on Software Engineering, vol. 186 (2003)
25. Yang, C.C., Chen, C.K., Chang, Y.H., Chung, K.H., Lee, J.K.: Streaming support for Java RMI in distributed environments. In: Proceedings of the 4th international symposium on Principles and practice of programming in Java, pp. 53–61 (2006)
26. Zhang, C., Jacobsen, H.: Refactoring middleware with aspects. IEEE Transactions on Parallel and Distributed Systems 14(11), 1058–1073 (2003)