An Extensible Framework for Annotation-based Parameter Passing in Distributed Object Systems

Sriram Gopal

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Eli Tilevich, Chair

Godmar Back

Naren Ramakrishnan

June 6, 2008

Blacksburg, Virginia

Keywords: Extensible Middleware, Metadata, Aspect Oriented Programming (AOP),

Generative Programming, Declarative Programming

Copyright 2008, Sriram Gopal

An Extensible Framework for Annotation-based Parameter Passing in Distributed Object Systems

Sriram Gopal

(ABSTRACT)

Modern distributed object systems pass remote parameters based on their runtime type. This design choice limits the expressiveness, readability, and maintainability of distributed applications. While a rich body of research is concerned with middleware extensibility, modern distributed object systems do not offer programming facilities to extend their remote parameter passing semantics. Thus, extending these semantics requires understanding and modifying the underlying middleware implementation.

This thesis addresses these design shortcomings by presenting (i) a declarative and extensible approach to remote parameter passing that decouples parameter passing from parameter types, and (ii) a plugin-based framework, *DeXteR*, that enables the programmer to extend the native set of remote parameter passing semantics, without having to understand or modify the underlying middleware implementation.

DeXteR treats remote parameter passing as a distributed cross-cutting concern. It uses generative and aspect-oriented techniques, enabling the implementation of different parameter passing semantics as reusable application-level plugins that work with application, system, and third-party library classes. The flexibility and expressiveness of the framework is validated by implementing several non-trivial parameter passing semantics as *DeXteR* plugins.

The material presented in this thesis has been accepted for publication at the ACM/USENIX Middleware 2008 conference.

Acknowledgements

My sincere thanks are due to many individuals, without whose assistance this work might not have been possible.

I am very grateful to my advisor and committee chair, Dr. Eli Tilevich, for his cheerful motivation and constant guidance. Dr. Tilevich has stood by me throughout the course of this project and has helped me overcome the research obstacles I confronted. The hours we spent discussing things including but not restricted to research problems, career paths and day-to-day world affairs have been truly insightful and memorable. I thank him for being a wonderful advisor.

I was fortunate to have Dr. Godmar Back and Dr. Naren Ramakrishnan on my thesis committee. I thank them for taking time out of their busy schedule for reviewing my thesis and providing valuable feedback.

I thank the graduate program committee for giving me the opportunity to pursue my graduate study. I must also thank Dr. Layne Watson and the Digital Library & Archives of University Libraries for providing me with the financial support during the first and the

second years of my Master's respectively.

I thank the members of my lab and all my colleagues for engendering an intellectual and a stimulating environment. Specifically, I would like to acknowledge the contributions of my colleagues Wesley Tansey and Gokulnath C. Kannan. Wesley's ideas during the initial phases of this project helped us shape our framework design. Gokulnath helped us evaluate our framework by developing a few plugins.

I thank all my friends for supporting me and making my stay at Virginia Tech a memorable one.

Most importantly, I thank my parents for their everlasting love and support.

Contents

Chapter 1: Introduction **1**

1.1 Overview 1

1.2 Thesis Statement 3

1.3 Contributions 4

1.4 Outline 5

Chapter 2: Background and Motivation **6**

2.1 Background 6

2.1.1 The Concept of Object Request Broker (ORB) 7

2.1.2 Java RMI Overview 8

2.1.3 Passing Parameters in Java RMI 10

2.1.4 Other Mechanisms 12

2.2 Motivation 14

2.2.1 A Motivating Example 14

2.2.2 Problems 17

Chapter 3: The *DeXteR* Framework **19**

3.1 Framework Overview 21

3.2 Framework API 22

3.3 Implementation Details 25

3.3.1 Compile-time 26

3.3.2 Load-time 26

3.3.3 Runtime 26

3.4 Bioinformatics Example Revisited 27

Chapter 4: Supporting Parameter Passing Semantics **29**

4.1 Lazy Semantics 30

4.2 Copy Restore Semantics 35

4.3 Copy Restore With Delta Semantics 37

4.3.1 Creating Linear Map 38

4.3.2 Calculating Delta 39

4.3.3 Restoring Changes 40

4.4 Streaming Semantics 41

4.5 Caching Semantics 47

4.6 Other Semantics 49

Chapter 5: Discussion **50**

5.1 Design Advantages 50

5.2 Design Constraints 53

Chapter 6: Related Work **55**

6.1 Separation of Concerns 55

6.2 Remote Parameter Passing 58

6.3 Other Systems 62

Chapter 7: Future Work and Conclusions **65**

7.1 Future Work 65

7.2 Conclusions 67

Bibliography **76**

Chapter A: Performance **77**

List of Figures

2.1 Object Request Broker. 7

2.2 Pass by *copy* creates an isomorphic copy. 11

2.3 Pass by *remote-reference* delegates calls back to the client. 11

3.1 Framework interceptor and plugin interaction. 22

3.2 Development and deployment process using DeXteR. 23

4.1 Lazy semantics plugin interaction diagram 31

4.2 Copy-restore semantics plugin interaction diagram 36

4.3 *Copy-restore with delta* algorithm 38

4.4 *Copy-restore with delta* algorithm by example 42

4.5 Performance gain of copy-restore with delta over copy-restore 43

4.6 Streaming semantics plugin interaction diagram 44

4.7 Caching semantics plugin interaction diagram 48

A.1 Pass by copy benchmark. 78

A.2 Pass by remote-reference benchmark. 79

List of Tables

5.1 Analysis of Java 6 JDK's public member fields (some overlap exists due to `Exception` classes spanning multiple packages). 54

Chapter 1

Introduction

1.1 Overview

Distributed Object Computing (DOC) middleware represents a significant component of modern distributed systems and has consequently become an integral part of modern software development. A distributed object system builds upon a programming language's object system to facilitate distributed application development.

One of the foremost modern distributed object systems, Java Remote Method Invocation [Sun97a], follows the design philosophy outlined in *A Note on Distributed Computing* [KWWW94]. Java RMI minimizes the complexity of the clients and the servers by retaining the semantics of the Java object model. At the same time, RMI makes use of different programming idioms for distributed computing in order to accommodate for the

differences in latency, calling semantics, and the possibility of partial failure, thereby making them apparent to the programmer. The success of Java RMI has influenced the design of other distributed object systems such as .NET Remoting [OH01].

One facet in which Java RMI differs from the Java object model is parameter passing. In Java RMI, while parameters of primitive Java types are always passed by *copy*, reference parameters are passed based on their runtime type. If an object's runtime type implements a `Serializable` interface, it is passed by *copy*; if it implements a `Remote` interface, it is passed by *remote-reference*.

This remote parameter passing design of Java RMI, however, imposes several limitations on the programmer. First, it assumes that all instances of the same type will be passed identically, thus restricting expressiveness. Second, remote method declarations do not reveal any details about how parameters are passed, thus forcing the programmer to examine each parameter type individually. This reduces readability and hinders program understanding. Finally, an existing class may have to be modified to implement a remote interface before its instances can be passed as parameters to a remote method, thus complicating maintainability. Specifically, in the case of third-party libraries, source code may be difficult or even impossible to modify. Further, though several novel, advanced remote parameter passing semantics such as *copy-restore* [TS08], *lazy* [Eug03], *streaming* [YCC+06], *caching* [ET01] have been proposed, the ability to incorporate these semantics requires that the programmer understand and modify the underlying middleware implementation.

This thesis addresses the afore-mentioned shortcomings of such a type-based remote param-

eter passing model. While our subsequent discussion uses Java RMI as an example, the insights are also applicable to other object-oriented languages and their distributed object systems.

1.2 Thesis Statement

"Decoupling parameter passing semantics from parameter types improves the expressiveness, readability, and maintainability of distributed applications. Further, remote parameter passing can be treated as a distributed cross-cutting concern, so that the native set of parameter passing semantics can be extended without having to understand or modify the underlying middleware implementation."

Cross-cutting concerns in software development represent functionalities or features that affect other features and often times result in scattered or entangled code. *Separation of concerns* is a guiding software design principle that helps decompose concerns ensuring robustness, modularity, reusability and maintainability. Several prior approaches have advocated treating orthogonal services such as logging, security as cross-cutting concerns for various software engineering benefits. This work takes a different direction by treating one of the core facets of a distributed object model, its parameter passing, as a distributed cross-cutting concern.

This research proves the thesis by developing a plug-in based framework, *DeXteR*, that enables an annotation-based parameter passing model and by implementing several non-trivial

remote parameter passing semantics as *DeXteR* plugins. Our technique uses a combination of generative and aspect-oriented programming techniques to transform a type-based remote parameter passing model to an annotation-based model transparently, and to enable third-party vendors or in-house programmers to seamlessly extend a native set of remote parameter passing semantics with additional semantics in the application space, without modifying the JVM or its runtime classes. The proposed approach is equally applicable to system classes, application classes and third-party libraries, and incurs negligible performance overhead.

1.3 Contributions

The technical material presented in this thesis 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 annotation-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.

1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 provides an overview of existing mechanisms for remote parameter passing in our example domain and presents the motivation behind the problem of type-based remote parameter passing in distributed object systems. Chapter 3 describes the design of our extensible framework that transforms the type-based parameter passing model to a declaration-based model and simplifies the creation of additional parameter passing semantics. Chapter 4 describes how we used our framework to add several non-trivial parameter passing semantics to RMI. Chapter 5 discusses the advantages and constraints of our approach. Chapter 6 discusses related work. Finally, Chapter 7 outlines future work directions and conclusions.

Chapter 2

Background and Motivation

This chapter provides background information and presents the motivation to the problem of type-based parameter passing in distributed object systems.

2.1 Background

To make arguments for or against any particular parameter passing model, we provide a general overview of our example domain. We chose Java RMI, as Java is one of the foremost languages for enterprise computing, with millions of developers worldwide. In our overview, we focus on the features of Java RMI that are most pertinent to the discussion and elide less directly-related details.

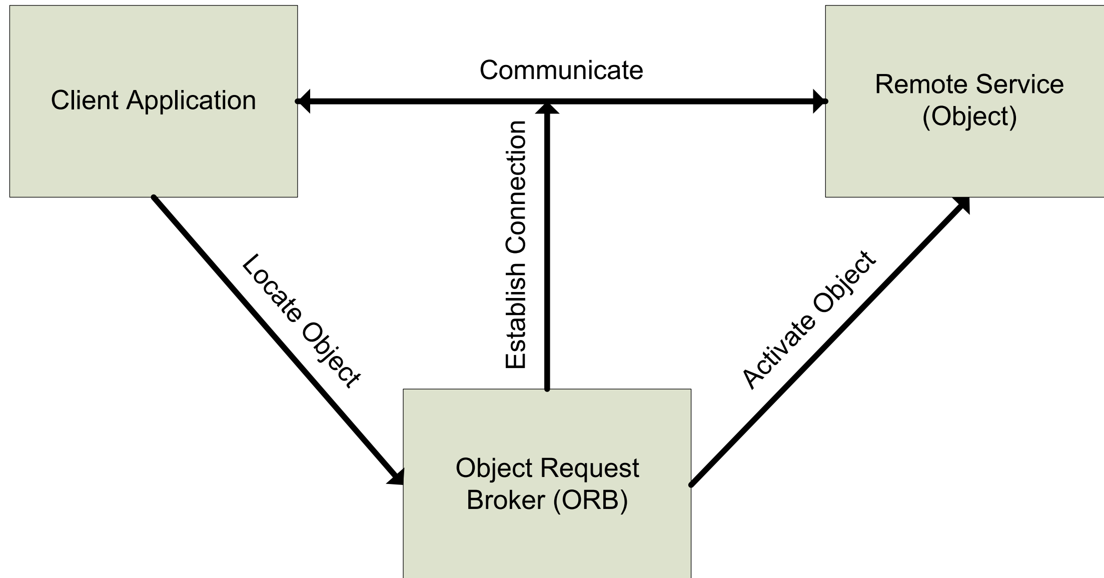

Figure 2.1: Object Request Broker.

2.1.1 The Concept of Object Request Broker (ORB)

In distributed computing, an Object Request Broker (ORB) [orb07] represents a piece of middleware technology that enables communication between distributed objects. With *interoperability* being one of its primary aims, the ORB enables piecing together of objects across vendor, software and machine boundaries.

The ORB allows objects to hide their implementation details from their clients. This transparency includes the programming language, the operating system, the underlying hardware and the object location. There are several variations of ORB technology such as CORBA [Gro98b], DCOM [BK98], Java RMI [Sun97a] etc. and each of these provide different levels of distribution transparency. The key idea behind an ORB is depicted in figure 2.1. The ORB provides a directory of services and helps the clients establish connections

with these services.

2.1.2 Java RMI Overview

The Java Remote Method Invocation (RMI) provides ORB-like capabilities as a native extension to Java. It is a distributed object system for executing the methods of a remote object on a different Java Virtual Machine (JVM). Similar to distributed systems that are based on the Remote Procedure Call (RPC) model, Java RMI exposes each remote method as if it were "a perfectly normal local call" [BN84]. However, it does not meet this objective entirely, as doing so is not only infeasible but also undesirable because distributed programming models require that the programmer be aware of latency, differences in calling semantics, concurrency and partial failure [KWWW94].

To be accessed from another JVM, a remote object's class must implement an interface that declares its remote methods and extends `java.rmi.Remote`. Furthermore, a remote object must publish its interface using `UnicastRemoteObject`. This functionality can be accessed either by extending this class or calling its static method `exportObject`. Publishing a remote interface is accomplished through the use of stubs.

The role of a stub is to redirect method invocations to the original remote object. Stubs and their corresponding remote objects implement the same set of remote interfaces. However, stubs extend a system RMI class (`RemoteStub`) and, as such, are not subclasses of their remote objects. An RMI stub can be generated either by using the RMI compiler, *rmic*,

or by using the dynamic proxy generator at runtime. Despite the dynamic proxy option, *rmic* is by no means obsolete. For example, remote objects that are not subclasses of `UnicastRemoteObject` can only use the *rmic* option.

Thus, when a client requests a reference to a remote object that implements an interface, RMI substitutes and returns an instance of the stub. From the client's perspective, making a remote method invocation on this reference is similar to calling an interface method locally. However, this invocation is actually made on the stub, which forwards the call to the remote object in the server VM. As a simple example, consider a `Remote` interface `RI` and a remote object `RO` implementing it:

```
1 interface RI extends java.rmi.Remote {
2   int foo() throws RemoteException;
3 }
4
5 class RO implements RI {
6   int foo() throws RemoteException { return 1; }
7 }
```

An RMI client can invoke remote methods through the `RI` remote interface as follows (low-level details such as exception handling are omitted):

```
1 //lookup RMI stub implementing RI in the registry
2 RI ri = (RI)Naming.lookup(‘‘url’’);
3
```

```
4 //remote invocations look exactly
5 //like local invocations
6 int i = ri.foo();
```

2.1.3 Passing Parameters in Java RMI

One facet of Java RMI that does not mimic local method calls is parameter passing. Because of the lack of a shared address space in a distributed object model, it would be impossible to emulate the local parameter passing mechanism of pass-by-reference for remote calls efficiently without modifications. Java RMI provides two natively supported mechanisms for remote parameter passing.

Pass By Copy. Pass by *copy* for a remote call approximates pass by *value* for a local call by creating a copy of an object passed as a parameter, as shown in figure 2.2. Changes to the copy are, therefore, not reflected on the original object. Java RMI supports this using object serialization [Sun97b], an application of the pickling technique [RWWB96]. Pickling is the process of creating a serialized representation of objects. By dening the serialized form to include meta information that identies the type of each object and the relationships between objects within a stream, pickling insures that the equivalent typed object and the objects to which it refers can be recreated. Java RMI preserves an object's state to a buffer using serialization. The serialized object can then be transferred to a remote network site, at which point the object's state is restored through deserialization and used as a parameter.

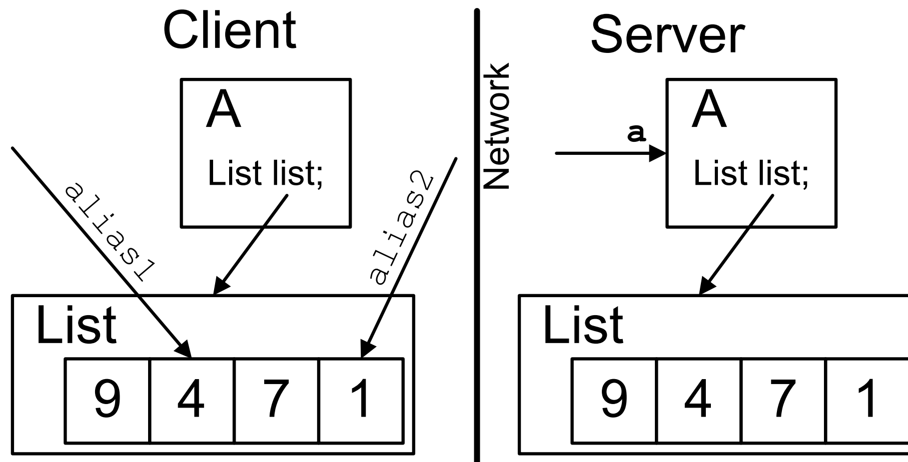

Figure 2.2: Pass by *copy* creates an isomorphic copy.

Pass By Remote-Reference. Pass by *remote-reference* for a remote call approximates pass by *reference* for a local call by passing a stub object that propagates all method calls to the original object, as shown in Figure 2.3. Pass by *remote-reference* addresses the need to reference an object over the network (e.g., when copying an object is prohibitively expensive or undesirable for security reasons).

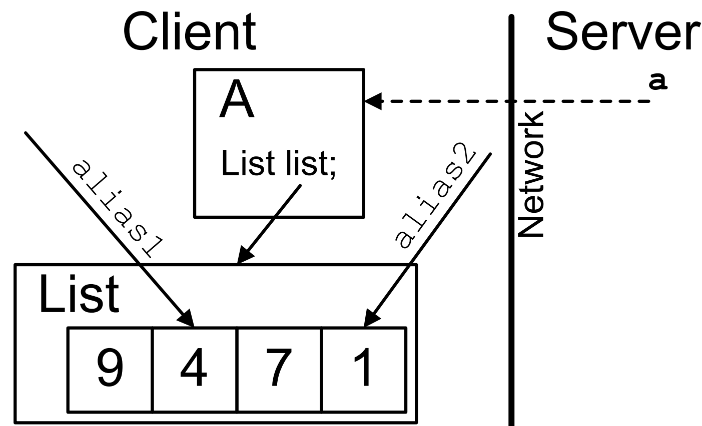

Figure 2.3: Pass by *remote-reference* delegates calls back to the client.

2.1.4 Other Mechanisms

In addition to the natively supported mechanisms of Java RMI, we review several other mechanisms for passing remote parameters that have been proposed in the literature.

Pass By Copy-Restore. Pass by *copy-restore* has been proposed as a middle ground between pass by *copy* and pass by *remote-reference*. Pass by *copy-restore* works by copying an object to the callee and then restoring the original object at the caller's side in place. Pass by *copy-restore* is not part of standard Java RMI, but can be implemented efficiently without changes to the Java language [TS08].

Lazy Parameter Passing. Lazy parameter passing [Eug03] a.k.a *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.*

Streaming Parameters. Passing objects by *streaming* could be used when parameters or return types are large objects such as multimedia. It involves buffering the large objects in the background without blocking the call. One such streaming methodology is described in [YCC+06], which presents a software architecture for supporting streaming in RMI, and

employs sophisticated pushing and aggregation mechanisms for obtaining large objects from multiple servers.

Parameter Substitution a.k.a Caching. *Caching* represents a useful mechanism for wide area networks wherein latency is one of the major concerns. It works by saving a copy of the state of parameter objects on the receiving node and using them for subsequent invocations without requiring a retransmission. This mechanism involves taking additional factors such as caching cost and consistency into consideration. [ET01] discusses some of these factors, and presents different caching strategies and consistency protocols for Java RMI.

Pass By Move. Pass by *move* [BHJL07] moves an object permanently to the callee, changing all local references to the object on the caller's side to be remote references. Such functionality requires sophisticated runtime system support [Dah00] or a significant rewrite of the original program (e.g., to change all direct references to indirect ones [TS02]).

Adaptive Parameter Passing. Sophisticated application programmers know better how a remote object is to be passed in their application. An interesting mechanism based on this idea is *adaptive parameter passing* [Lop96]. This mechanism enables passing a subset of an object's state by *copy*. It provides linguistic and runtime support for traversing a parameter's object graph and selecting a subset for copying to the remote method.

2.2 Motivation

Despite widespread use, the remote parameter passing model of RMI has some serious shortcomings that adversely affect the development, understanding, and maintenance of distributed applications.

2.2.1 A Motivating Example

Consider the task of leveraging idle computing resources for distributed scientific computation. Organizations have hundreds of workstations connected to local area networks (LANs) that stay unused for hours at a time. We would like to set up an ad-hoc grid that will use the idle workstations to solve bioinformatics problems. Specifically, 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.

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 computationally-intensive methods: `align`, `cross`, and `mutate`.

```
1 class Sequence {...}
```

```
2
3 class SequenceDB {
4   void append(Sequence s) {...}
5   boolean isFull() {...}
6 }
7
8 interface WorkerInterface {
9   void align(SequenceDB allSeqs, SequenceDB candidates, Sequence toMatch);
10  Sequence cross(Sequence s1, Sequence s2);
11  void mutate(SequenceDB seqs);
12 }
13
14 class Worker implements WorkerInterface {
15   void align(SequenceDB allSeqs, SequenceDB candidates, Sequence toMatch) {
16     for (Sequence s : candidates)
17       if (!allSeqs.isFull() && satisfiesThreshold(s))
18         allSeqs.append(s);
19   }
20
21   Sequence cross(Sequence s1, Sequence s2) {
22     return doCross(s1, s2);
23   }
24
25   void mutate(SequenceDB seqs) {
26     for (Sequence s : seqs.getSequences())
```

```
27        doMutate(s);
28    }
29
30    ...
31 }
```

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.

2.2.2 Problems

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 can pass it by *copy*, so that its contents can be examined and compared efficiently. However, 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 may have to be changed as well. Passing `allSeqs` by *remote-reference* requires the type of `allSeqs` to become a remote interface. Further, 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 [DLFM96] 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 resulting from frequent updates to the `seqs` 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 [TS08].

Java RMI however, does not natively support *copy-restore*. More importantly, it lacks the design flexibility for supporting such parameter passing extensions. One could use a custom implementation provided either by a third-party vendor or an in-house expert programmer. However, this requires the third-party developer to have a detailed understanding of the RMI implementation in order to modify it to include *copy-restore* support. Further, in order to use this custom implementation of *copy-restore*, one needs to have 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.

Chapter 3

The *DeXteR* Framework

To overcome limitations of RMI remote parameter passing model, which is type-based and inextensible, we present an alternative model. Our model is *annotation-based* and extensible. It makes remote parameter passing resemble that of local parameter passing in mainstream programming languages. Specifically, a passing mechanism for each parameter is specified at remote method declarations. Decoupling parameter passing from parameter types, increases expressiveness, improves readability, and eases maintainability.

The parameter passing design of mainstream programming languages offers valuable lessons. Languages such as C, C++, and C# express the choice of parameter passing mechanisms through method declarations with special syntactic 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 follow to a similar approach for remote method calls, but one that is designed for distributed communication.

Recognizing that many existing distributed applications are built upon a type-based model, this chapter presents a technique for transforming a type-based remote parameter passing model to use an annotation-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 show that an annotation-based model to remote parameter passing simplifies adding new semantics to an existing distributed object model. Specifically, we present an extensible plug-in-based framework, **DeXteR** (**D**eclarative **Exte**nsible **R**emote Parameter Passing), 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 extensions in the application space, without modifying the JVM or its runtime classes.

The chapter begins by giving a general overview of DeXteR, followed by a description of the API it provides, the implementation details and how the example bioinformatics application presented in Chapter 2 can be distributed with ease using the annotation-based approach supported by DeXteR.

3.1 Framework Overview

DeXteR implements annotation-based remote parameter passing 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 semantics is an independent plugin component.

DeXteR uses the Interceptor Pattern [SRSS00] to expose the invocation context explicitly on the client and the server sites. The Interceptor pattern captures techniques for extending the functionality of a complex system at specific interception points. While Interceptors have been used in several prior systems [FR03] 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 3.2 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 plugin-specific code generators. The framework-specific code generator synthesizes the code for the client and the server interceptors using aspects. The plugin-specific code generators synthesize the code pertaining to the translation strategy for supporting a specific parameter

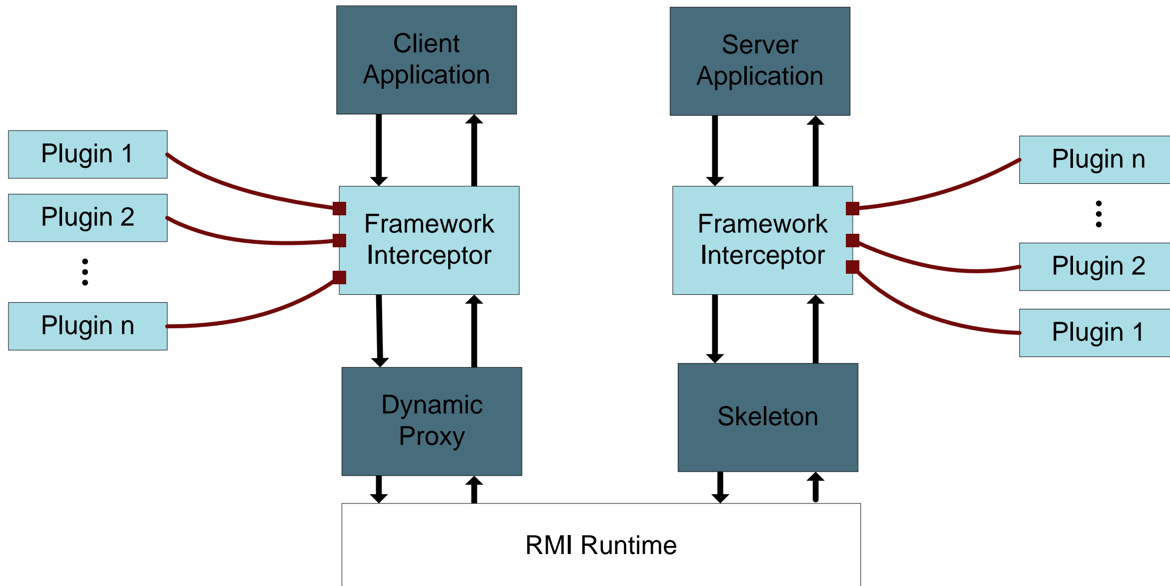

Figure 3.1: Framework interceptor and plugin interaction.

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.

3.2 Framework API

DeXteR provides extension points for parameter passing plugins in the form of the `IGenerator` interface and the `InterceptionPoint` interface. Developing a new plugin involves implementing the `InterceptionPoint` interface and the optional `IGenerator` interface, iden-

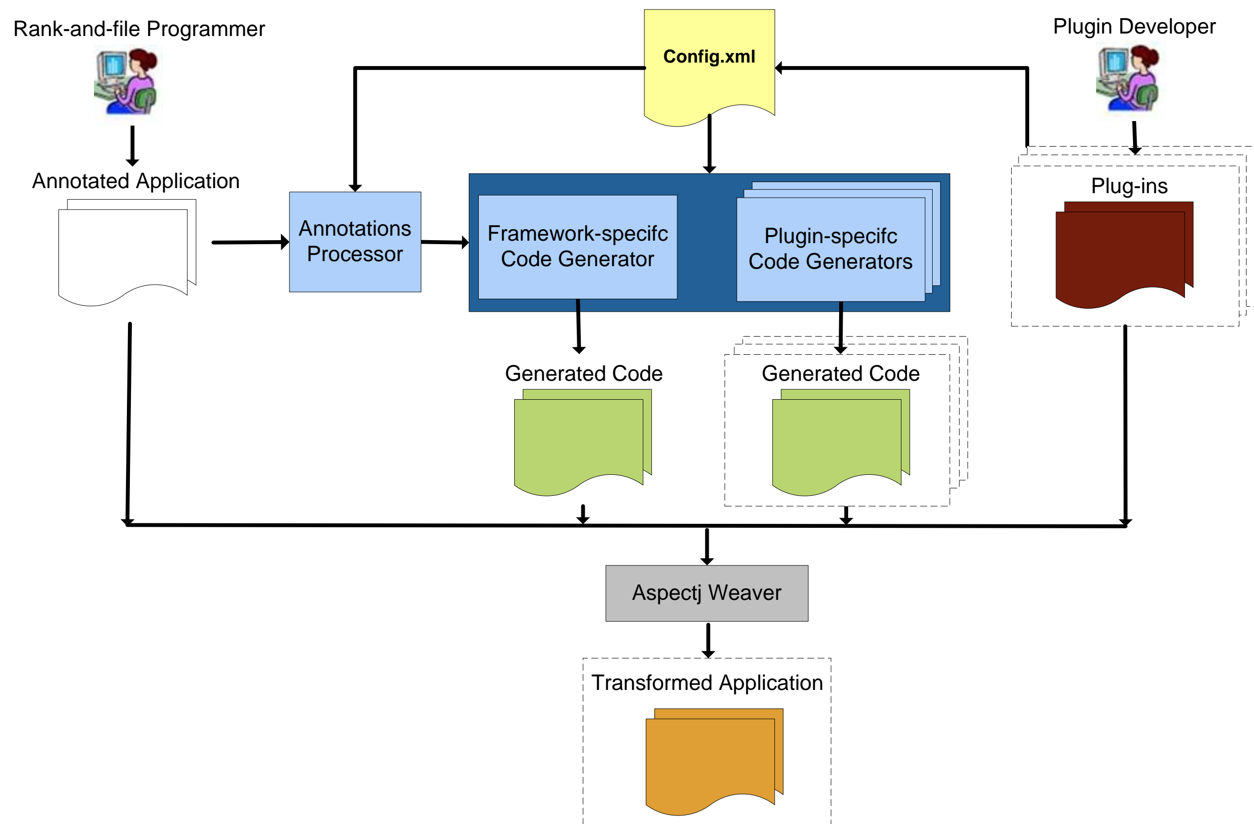

Figure 3.2: Development and deployment process using DeXteR.

tifying the interception points of interest, providing the functionality at these interception points, and registering the plugin with the framework.

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

The `IGenerator` interface forms the compile-time part of a plugin. During compile-time, DeXteR exposes the annotation information extracted from the RMI application to the respective parameter passing plugins. Plugins can use this information to generate code, which can then be used at run-time for implementing a specific parameter passing strategy.

```
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);
```

```
11      Object retAfterServerCall(Object target, Object ret);
12      Object[] customRetAfterServerCall(Object target);
13  }
```

The `InterceptionPoint` interface forms the run-time part of a plugin. It 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. Plugins can thus modify the invocation arguments using the code generated at compile-time. In addition, DeXteR enables sending custom information between the client- and the server-side plugins. This custom information is simply piggybacked to the original invocation context.

3.3 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.

3.3.1 Compile-time

For each remote method, DeXteR generates AspectJ code that injects a wrapper method into the remote interface and the server implementation using *inter-type declarations*, which enable introducing new members. In addition, DeXteR pointcuts on the execution of that method in the stub (i.e., implemented as a dynamic proxy) to provide a wrapper. This is accomplished by providing an *around* advice, which runs in place of a specific execution point. All the AspectJ code that provides the interception functionality is automatically generated at compile time, based on the remote method's signature.

3.3.2 Load-time

The generated aspects are weaved into the stub (i.e., dynamic proxy) on the client side, and the remote interface and server implementation on the server side when these classes are loaded into the virtual machine.

3.3.3 Runtime

At runtime, 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.

3.4 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 Chapter 2 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.

```
1  interface WorkerInterface extends Remote
2  {
3     void align(@RemoteRef SequenceDB matchingSeqs, @Copy SequenceDB candidates,
4                @Copy Sequence toMatch) throws RemoteException;
5     @Copy Sequence cross(@Copy Sequence s1, @Copy Sequence s2) throws RemoteException;
6     void mutate(@CopyRes SequenceDB seqs) throws RemoteException;
7  }
8
9  class Worker implements WorkerInterface
10 {
11    void align (@RemoteRef SequenceDB matchingSeqs,
12                @Copy SequenceDB candidates,
13                @Copy Sequence toMatch) throws RemoteException { ... }
```

```
14
15    @Copy Sequence cross(@Copy Sequence s1, @Copy Sequence s2)
16                          throws RemoteException { ... }
17
18    void mutate (@CopyRes SequenceDB seqs)
19                          throws RemoteException { ... }
20 }
```

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). This requirement however, must not impose any additional burden on the programmer, as a modern IDE such as Eclipse [Fou07], NetBeans [Mic07], or Visual Studio [Cor07] can be made to reproduce these annotations when providing method stub implementations for remote interfaces.

Chapter 4

Supporting Parameter Passing Semantics

We validate the expressiveness of our framework by extending the set of available parameter passing semantics of RMI with several non-trivial state-of-the-art semantics, introduced earlier in the literature [TS08, Eug03, YCC$^+$06, ET01]. This chapter describes the strategies for implementing these parameter passing mechanisms as DeXteR plugins.

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

them using DeXteR.

One of the new semantics we present in this chapter is an optimization of the algorithm for *copy-restore* [TS08]. 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. Our extensible framework makes it possible to use these different versions of the *copy-restore* algorithm for different remote calls in the same application.

4.1 Lazy Semantics

Lazy parameter passing [Eug03], 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 type `A` by *lazy* semantics does not require defining any special interface nor `A` implementing one. Instead, the plugin-specific code gen-

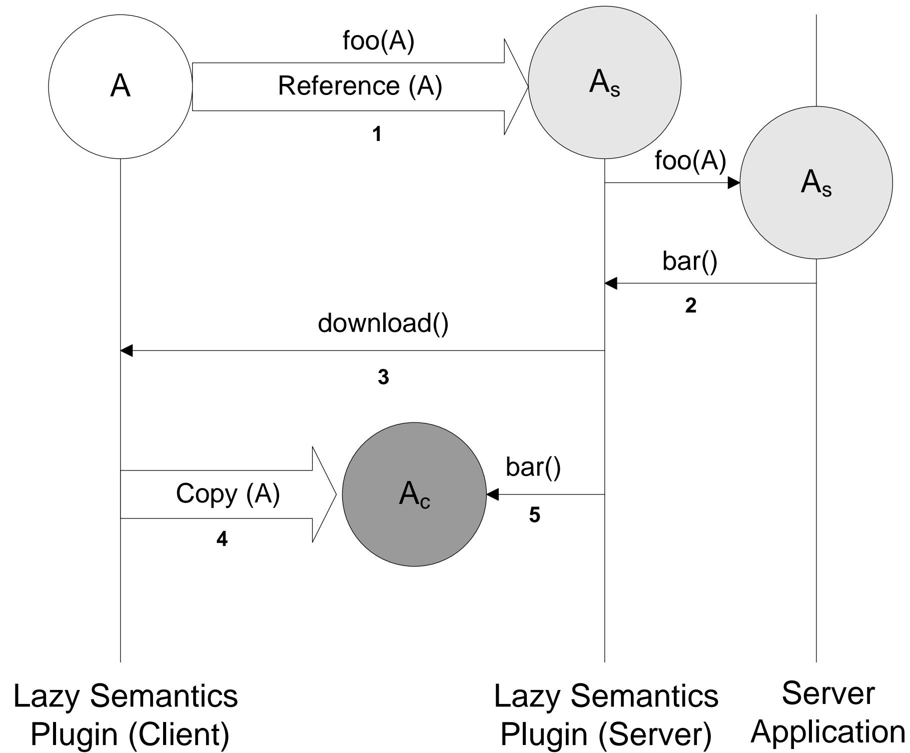

Figure 4.1: Lazy semantics plugin interaction diagram

(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.)

erator 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 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 [Eug06], 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`.

```
1 class A {
2   public void bar() {...}
3 }
4
5 // Generated remote interface
6 interface AIface extends Remote {
7    public void bar() throws RemoteException;
8    public A download() throws RemoteException;
```

[1]Replacing direct fields accesses with methods has become such a common transformation that AspectJ [KHH+01] provides special fields access pointcuts (i.e., `set`, `get`) to support it.

```
9  }
10
11 // Generated client proxy
12 class A_DynamicProxy implements AIface {
13    // delegate remote object
14    private A remoteParameter;
15
16    public A download() {
17       // serialize remoteParameter
18    }
19
20    public void bar() throws RemoteException { ... }
21 }
22
23 // Generated server proxy
24 class A_ServerProxy extends A {
25    // lazy copy of the remote object
26    private A a;
27    // type-incompatible stub
28    private AIface stub;
29
30    public A_ServerProxy(AIface stub) {
```

```
31          this.stub = stub;
32      }
33
34      synchronized void download() {
35          // Obtain a copy of the remote parameter
36          a = stub.download();
37      }
38
39      public void bar() {
40          // Dereference the stub
41          stub.download();
42          // Invoke the method on the copy
43          a.bar();
44      }
45  }
```

Any invocation made on the parameter (i.e., server proxy) by the server results in a call to its `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 by the server proxy. 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*.

4.2 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

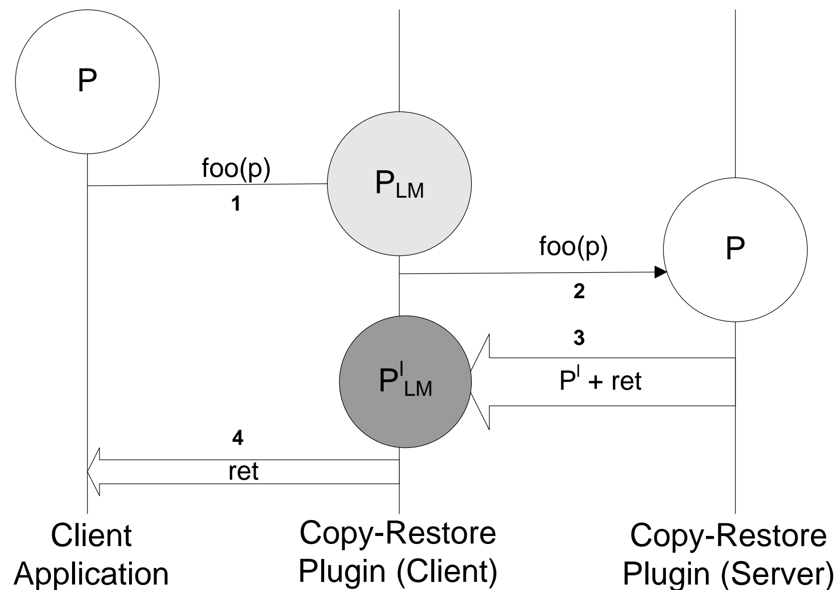

Figure 4.2: Copy-restore semantics plugin interaction diagram

(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.)

restoring the changes made by the server after the call. The task is simplified by the well-defined hook points provided by the framework. The *copy-restore* plugin obtains a copy of the parameter `A` and creates a linear map of all objects reachable from the parameter on both the client and the server sites prior to the invocation. 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 in the form of a linear map. This is accomplished using the custom information passing facility provided by the framework. The client-side plugin obtains the linearmap from its server-side peer, compares it with the linearmap it holds and restores the changes to the original parameter A in the client's JVM.

4.3 Copy Restore With Delta Semantics

For single-threaded clients and stateless servers, *copy-restore* [TS08] makes remote calls indistinguishable from local calls, as far as reference parameter passing is concerned. However, in a low bandwidth high latency networking environment, such as in a typical wireless network, the *copy-restore* implementation may be inefficient. The inefficiency is due to 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 ability to use such an optimized *copy-restore* implementation again presents a compelling case for extensibility and flexibility in remote parameter passing.

The pseudocode for the optimized *copy-restore* algorithm, which we term as *copy restore with delta* is described in figure 4.3.

1. *Create a linear map of all argument reachable objects*
2. *Serialize a deep copy of the linear map to the server*
3. *On the server, create two deep copies of the deserialized linear map (say Lmap1 and Lmap2)*
4. *Execute the remote procedure with Lmap1 as the argument.*
5. *Serialize a deep copy of Lmap1. By matching up Lmap1 and Lmap2,*
   i. *Replace every old object with a handle encoding any changes resulting from the remote call*
   ii. *Serialize every new object as is*
6. *On the client, replace every handle deserialized with the corresponding old objects from the client's linear map and replay the encoded changes*

Figure 4.3: *Copy-restore with delta* algorithm

4.3.1 Creating Linear Map

The linear map of all objects reachable from the reference argument is constructed during serialization. The linear map data structure contains references to all the argument reachable objects in the serialization traversal order. To ensure that the referents of the linear map are not stopped from being reclaimed, the linear map uses weak references.

4.3.2 Calculating Delta

Prior to invoking the method, two linear maps (`Lmap1` and `Lmap2`) and a mapping of corresponding references in these linear maps using reference equality are constructed on the server. Once the call completes with the referents of `Lmap1` as argument and the remote method mutates its argument, the changes are sent back to the client. This involves calculating the delta between the original object `Lmap1` and the callee modified object `Lmap2` and encoding the changes that are then reproducible by the client on its data structure. A traversal of `Lmap1` is performed and each object in `Lmap1` with a corresponding old object in `Lmap2` is replaced by a handle.

The simplified handle format is shown below. The identifier `id` indicates the position of the old object in the original 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 and is an `ArrayList` of type long. For primitive fields, this value represents the modified value and for object fields, this value represents the position in `chObject`, an `ArrayList` of objects, which contains the modified references.

```
1 class Handle{
2    int id;
3    ArrayList<Long> chId;
4    ArrayList<Long> chScript;
5    ArrayList<Object> chObject;
6    ...
```

```
7 }
```

4.3.3 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, the unused references are reclaimed using garbage collection.

To illustrate our algorithm with an example, 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 are aliased by non-tree pointers `alias1` and `alias2`. Consider a remote method such as the one show below, for which tree `t` is passed as a parameter.

```
1 void alterTree (Tree tree) {
2    tree.left.data = 0;
3    tree.right.data = 9;
4    tree.right.right.data = 8;
5    tree.left = null;
6    Tree temp = new Tree (2, tree.right.right, null);
7    tree.right.right = null;
8    tree.right = temp;
9 }
```

Figure 4.4 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 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 4.5 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.

4.4 Streaming Semantics

Passing objects by *streaming* is useful when parameters or return types are large objects. It involves buffering the large object in the background without blocking the call. Streaming differs from the lazy semantics in the way the copy of an object is obtained. Therefore, to support the streaming semantics, we follow a strategy similar to the lazy semantics. Since

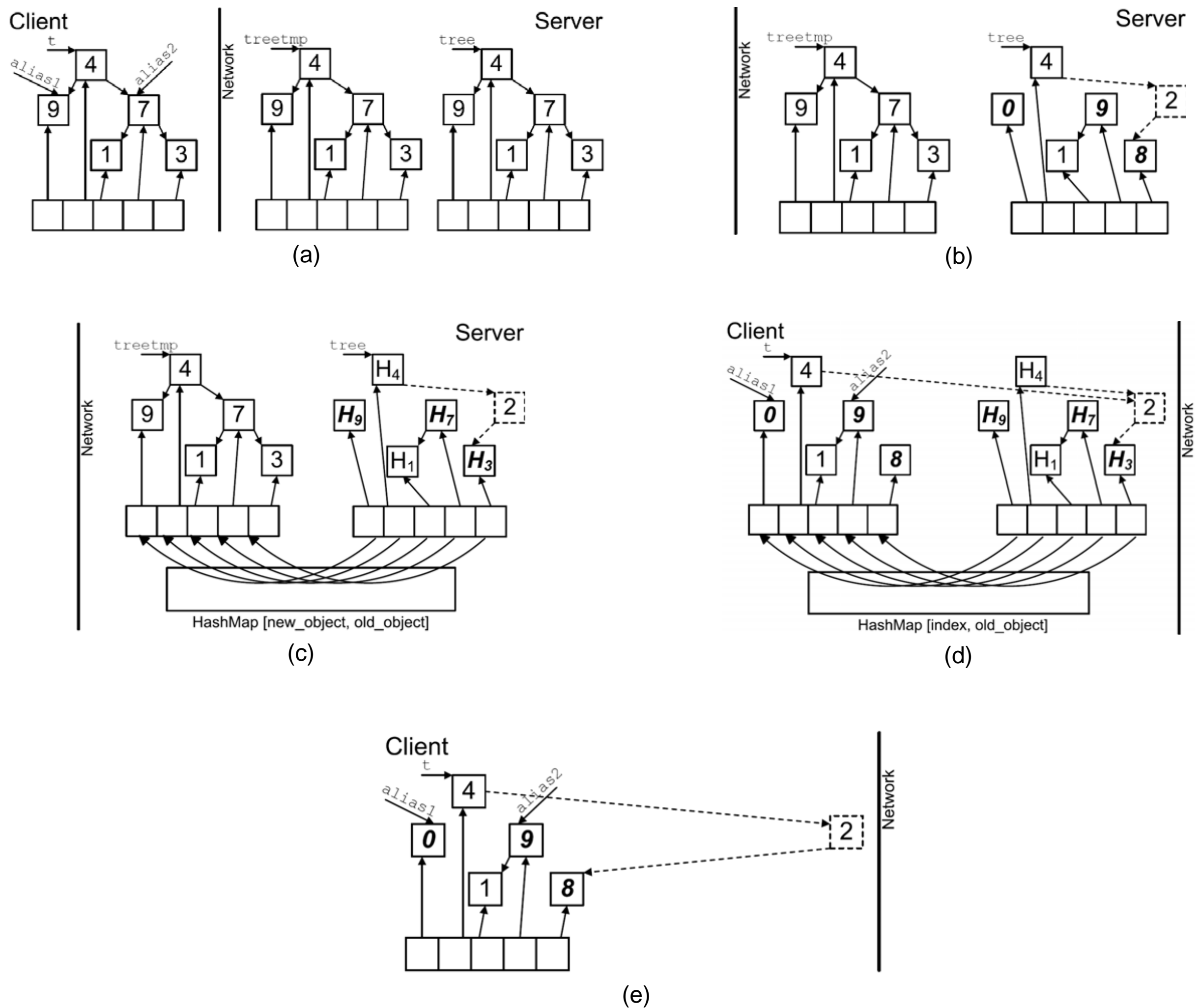

Figure 4.4: *Copy-restore with delta* algorithm by example (a) State after step 3. (b) State after step 4. The remote procedure modified the parameter. (c) State during step 5. Copy the modified objects (even those no longer reachable through `tree`) back to the client; compute the delta script for modified 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.

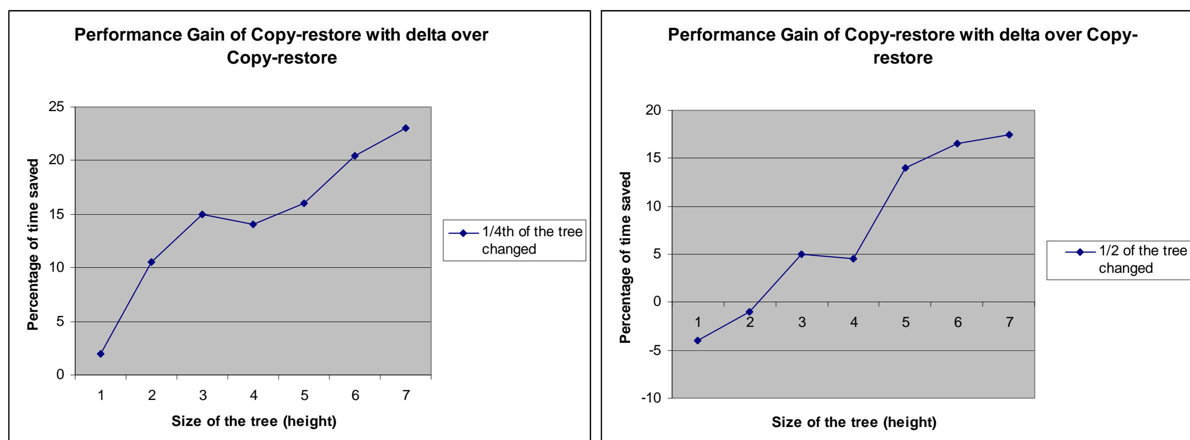

Figure 4.5: Performance gain of copy-restore with delta over copy-restore

streaming makes more sense for return types than parameters, our description below will focus on return types.

Our strategy for supporting streaming involves transmitting an object initially by *reference* by employing a pair of proxies (`A_DynamicProxy` and `A_ClientProxy`). We use the plugin-specific code generator to generate the proxies and the remote interface during compile time. Returning an object of type `A` will result in the server-side plugin replacing it with a type-incompatible stub `A_DynamicProxy`. The client-side plugin wraps this type-incompatible stub into a stub `A_ClientProxy` that is type-compatible with the return type of the remote method. Prior to returning the wrapped return type to the client, the client-side streaming plugin obtains a weak reference to it so that its referent is not prevented from being reclaimed and spawns a thread with the aim of populating it with a local copy of the returned object. The spawned thread invokes the `download()` method on the type-incompatible stub `A_DynamicProxy` enclosed within the type-compatible stub `A_ClientProxy` instance. The

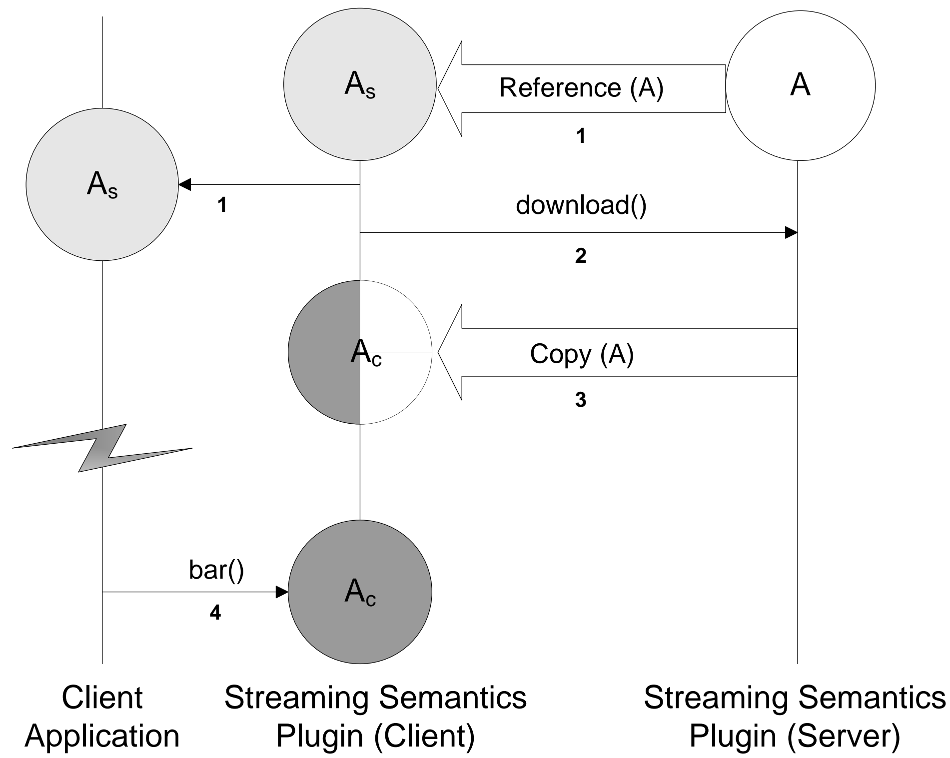

Figure 4.6: Streaming semantics plugin interaction diagram

(A: Serializable Object; A_s: Stub of A; A_c: Copy of A; (1) A is returned from server to client; (2) Client plugin spawns a thread and calls `download()` on server plugin; (3) Server plugin sends a copy of A, A_c and the client plugin starts buffering it; (4) Client calls `foo()` on the buffered A_c.)

`download()` method returns a copy of the `A`, which is populated within the `A_ClientProxy` instance by the client-side plugin using the weak reference it holds.

Future invocations made by the client on the return type are handled at the client end as soon as a copy of the entire object has been streamed. If not, the invocations are delegated to the server using the *remote-reference* held.

```
1  // Generated client proxy
2  class A_ClientProxy extends A
3  {
4     // streamed copy of the remote object
5     private A a;
6     // type-incompatible stub
7     private AIface stub;
8     // streaming completion indicator
9     private boolean isBuffered;
10
11    public A_ClientProxy(AIface stub) {
12       this.a = null;
13       this.stub = stub;
14       this.isBuffered = false;
15    }
16
17    public void deref() {
18       // obtain a copy of the remote parameter
19       a = stub.deref();
```

```
20     }
21
22     public void setBufferedStatus(){
23        isBuffered = true;
24     }
25
26     public boolean isBuffered(){
27        return isBuffered;
28     }
29
30     public void bar() {
31        if(isBuffered()) {
32           // invoke the method on the copy
33           a.bar();
34        }
35        else {
36           // invoke the method on the remote object
37           stub.bar();
38        }
39     }
40  }
```

The streaming mechanism described above could be viewed as a form of passing be *asynchronous copy*.

4.5 Caching Semantics

In order to demonstrate the expressive power of our framework, we also chose to implement a simplified form of *parameter substitution* a.k.a *caching* that follows a simple caching strategy and a basic consistency policy. A more comprehensive form of parameter substitution based on different consistency guarantees can be found at [ET01].

Parameter passing by *caching* can be used when unchanged resources/parameters need to be sent often from the client to server or vice-versa. It involves saving a copy of the state of parameter objects on the receiving node and using them for subsequent invocations without requiring a retransmission. *Caching* effects are more pronounced in case of WANs as cost of caching is worth the latency and bandwidth gains.

When a parameter is transmitted for the first time, the client-side caching plugin stores a copy of the parameter in its local cache prior to serializing a copy of it to the server using the pass by *copy* strategy outlined earlier. On obtaining the parameter object, the server-side caching plugin stores a copy of it in its local cache prior to providing the remote method with the parameter object. Since the plugin caches a deep copy of the parameter object, mutating the parameter object does not affect the state of the cached object.

Subsequent invocations involving the same parameter object state result in the client-side caching plugin substituting the original parameter object with an object identifier and sending it to the server. The server-side plugin does the reverse process. It uses the identifier sent by its client-side peer to retrieve the object state from its local cache and uses this cached

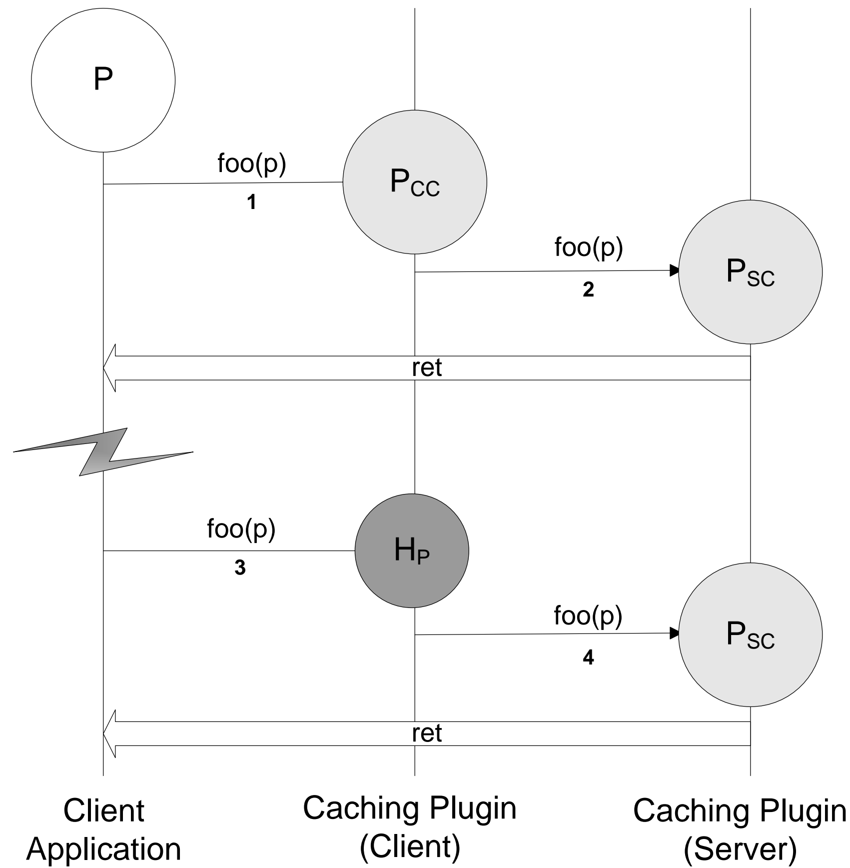

Figure 4.7: Caching semantics plugin interaction diagram

(P: Set of parameters passed to `foo`; P_{CC}: Parameter in client cache; P_{SC}: Parameter in server cache; H_P: Handle for parameter P; (1) The client invokes method `foo()` passing parameter P; (2) The client-side and the server-side plugins cache P as P_{CC} and P_{SC} respectively before invoking the original `foo(p)`; (3) The client makes a subsequent invocation of `foo()` with the same parameter P; (4) The client-side plugin replaces P with handle H_P and the server-side plugin retrieves P_{SC} corresponding to H_P and invokes `foo()` with P_{SC}.)

object as the parameter for the remote method.

4.6 Other Semantics

The advantages of DeXteR are in supporting a diverse set of remote parameter passing semantics through a uniform and an intuitive API.

DeXteR offers the advantage of supporting a wide variety of remote parameter passing semantics through a uniform API. By decoupling parameter passing from passing types, DeXteR provides the flexibility for including new parameter passing semantics as well as optimization strategies. Developments in hardware and software designs are likely to cause the creation of new parameter passing semantics and optimization mechanisms. These mechanisms 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 mechanisms 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.

Chapter 5

Discussion

This chapter discusses some of the advantages of the DeXteR framework as well as some of the constraints imposed by our design.

5.1 Design Advantages

Expressing remote parameter passing choices as a part of a method declaration has several advantages over a type-based system. Specifically, an annotation-based approach increases expressiveness, improves readability, and eases maintainability. To further illustrate the advantages of our annotation-based 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/or 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 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 to modify the semantics by simply specifying a different parameter passing annotation.

Extensibility. Even if the *copy-restore* semantics 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. As any new level of abstraction introduces some overhead, we had to ensure that the overhead imposed by DeXteR is not unreasonable. Since the latency of a remote call is orders of magnitude greater than that of a local call, the overhead of additional local calls added to the remote call by DeXteR is negligible. Our initial set of experiments confirm this fact, thereby demonstrating that our approach is feasible and the small overhead incurred due to DeXteR is worth the software engineering benefits. For further details, refer to appendix A.

5.2 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 an 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 [TS02]. 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 classes, it is not possible to support direct member field access for instances of system classes passed by *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 only ones used by application developers. As the table demonstrates, approximately 1% of classes contain non-final member fields. However, the

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

Table 5.1: Analysis of Java 6 JDK's public member fields (some overlap exists due to `Exception` classes spanning multiple packages).

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 more than 5,500 analyzed classes could potentially pose a problem, with two public member fields not accessible by getter methods.

Chapter 6

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.

6.1 Separation of Concerns

Several language-based and middleware-based approaches have been proposed for addressing the challenges in modeling cross-cutting concerns.

Aspect Oriented Programming. Aspect Oriented Programming (AOP) [KLM+97] provides mechanisms for modularizing a wide range of cross-cutting concerns. Several prior approaches have advocated using aspect-oriented techniques to improve various properties of middleware systems, with the primary focus on modularization [ZJ03, EM04]. AspectJ

[KHH+01], a popular aspect oriented extension to the Java language, provides hook patterns (*pointcuts*) for capturing a groups of events (*joinpoints*) and enables insertion of programming logic (*advice*) that can inspect and modify the data at these points. Unlike prior approaches, which use AOP techniques for modeling orthogonal cross-cutting concerns, our framework uses AOP techniques to separate one of the core facets of a distributed object system, its remote parameter passing model.

Distributed AOP. Approaches such as Java Aspect Components (JAC) [PSD+04], and DJCutter [NCT04] support distributed AOP. The JAC framework enables the advice to be added or removed dynamically. DJCutter, an extension to AspectJ, provides special language constructs for supporting *remote pointcuts*. Our framework could use these approaches as an alternative to AspectJ.

Feature Oriented Programming. Feature Oriented Programming (FOP) [Pre97] represents a paradigm for incremental software development. It involves decomposing a software system into features and providing incremental refinements to these features. However, unanticipated features and extensions may result in code tangling and code scattering. Thus, despite its support for evolvability, FOP cannot always effectively modularize cross-cutting concerns. Therefore, FOP techniques alone are unlikely to be sufficient as an alternative implementation mechanism.

Other Techniques. Proxies and Wrappers [FBF03, SM99] are commonly used patterns for introducing late bound cross-cutting features, though in an application-specific manner. Interceptors [SRSS00] are a common extensibility-enhancement patterns for transparent addition of services in complex software systems. Mixin Layers [SB98] is a layered approach for adding features to methods in different classes. Aspectual Mixin Layers (AMLs) [ALS06] take a middle-ground between AOP and FOP by providing an architectural integration of aspects and features for experiencing the best of both AOP and FOP in incremental development cycles. A clear exposition of compositional versus aspectual views of program evolution is presented in [HOT02]. Our framework's implementation combines some of these techniques, including proxies, wrappers, and interceptors.

DADO Framework. A closely related work is the DADO [WJD03] 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, while DADO could be used to implement remote parameter passing semantics, it would have to be supplemented with source or binary transformation facilities.

Flick Compiler. IDL-based systems separate interface definition from their language binding using an IDL compiler. However, they may be limited in flexibility if the language binding cannot be adapted when necessary. Typical IDL compilers are rigid and limited to supporting only a single IDL, a fixed mapping onto a target language, and a narrow range of data encodings and transport mechanisms. The Flick compiler [EFF+97] addresses the above limitations of a traditional IDL compiler. It provides flexibility in language bindings by separating the presentation from the interface in RPC and IDL [FHL94, FHL95]. The Flick compiler supports multiple IDLs, diverse data encodings, multiple transport mechanisms, and application of numerous optimizations to all of the code it generates.

SPOON Framework. The SPOON [Paw05] framework provides a program transformation tool that takes advantage of Java 5 annotations to define and parameterize user-defined transformations. Using compile-time reflection, SPOON enables annotation driven AOP with pure Java. Base programs can thus be annotated to define how and where the aspects are weaved. This can be used as an alternative to AspectJ in our implementation.

6.2 Remote Parameter Passing

IDL-based Systems. Multi-language distributed object systems such as CORBA [Gro98b], DCOM [BK98], 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. The IDL specification is translated into a conventional programming language such as C, C++ or Java. Traditional RPC systems thus, have separated the IDL and the target language mappings, for flexibility reasons.

The design of Java RMI, however, no longer distinguishes between a language-independent IDL specification and a mapping to concrete implementation language. Specifically, in RMI, Java interfaces have supplanted IDL specifications. Despite the simplicity advantages of this design, it lacks flexibility when it comes to remote parameter passing. Our framework, DeXteR, addresses this particular issue of RMI design.

In fact, DeXteR goes even beyond some IDL-based approaches that can be limited in flexibility if the language binding cannot be adapted as necessary [FHL94, FHL95]. Some IDL implementations do not completely decouple parameter passing semantics from parameter types. When the IDL interface is mapped to a concrete language, the generated implementation may still rely on a type-based parameter passing model of the target language. As an example, in mapping IDL to Java [Gro03], 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 [Gro98a].

.NET Remoting. .NET Remoting [OH01] 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 .NET Remoting contains some declarative elements in its parameter passing model, it has several shortcomings.

DOORASTHA System. Doorastha [Dah00] 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 our approach is influenced by the design of Doorastha, it differs from Doorastha in the following ways. First, Doorastha does not provide complete decoupling of parameter passing from the parameter types as it requires annotating classes of remote parameters with the desired passing style. Furthermore, it passes unannotated remote parameters 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 approach to remote parameter passing.

The Opentalk Communication Layer. The Opentalk communication layer [Inc02] consists of a set of frameworks and components, which provide a rich and extensible environment for development, deployment, maintenance, and monitoring of distributed Smalltalk applications. It has a complete request broker component implementation to provide transparent communication between Smalltalk images. The broker is created and configured prior to remote object communication. Like other distributed object systems, it supports pass by *value* and pass by *reference*. By default, all immediate objects (`nil`, `true`, `false`, `Characters` and `SmallIntegers`), `Magnitudes`, `ByteStrings`, `ByteSymbols`, some collections, and others are passed by value. Parameters that are complex objects, even though not exported, are exported automatically and passed by reference. The framework enables overriding this default behavior by forcing objects to be passed by value or by reference in a declarative style using keywords `asPassedByValue` and `asPassedByRef`. This approach however, does not follow a fully declarative style as the default behavior is still type-based. Furthermore, it does not support extending the native parameter passing modes to include newer ones.

6.3 Other Systems

KaRMI. Several systems improve the performance of RMI by using a more efficient serialization mechanism. KaRMI [PHN00] 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 [MvNV+99, MVNV+01] takes an alternative approach by using native code compilation to support compile and run time generation of marshaling code. Similar to standard Java RMI, our declaration-based approach relies heavily on serialization and will benefit from these optimizations.

Reflective RMI. Thiruvathukal et al. [TTK98] propose an alternative approach to implementing a remote procedure call mechanism for Java based on reflection. The approach employs the reflective capabilities of the Java language to invoke methods remotely. This simplifies the programming model, as a class does not have to be declared `Remote` for its instances to receive remote calls. This approach does not, however, aim at providing greater expressiveness to remote parameter passing, as their target domain is high-performance applications.

Distributed Shared Memory Systems. 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 [CBZ91], Orca [BBH+98], and, in the Java world, cJVM [AFT99], DISK [SM02], and Java/DSM [YC97]. DSM systems can be viewed as sophisticated implementations of pass by *remote-reference* semantics, to be contrasted with the standard RMI implementation. Nevertheless, the focus of DSM systems is very different than that of distributed object systems. DSMs are used when distributed computing is a means to achieve parallelism, providing correct and efficient semantics for multi-threaded execution. To achieve high performance, DSM systems create complex memory consistency models and require the programmer to implicitly specify the sharing properties of data. By contrast, our approach attempts to provide greater expressiveness for programming with a distributed object system.

Automatic Partitioning Tools. A special kind of tools that attempt to bridge the gap between DSMs and middleware are *automatic partitioning tools*. They split centralized programs into distinct parts, which run on different network sites. Thus, the objective of these automatic partitioning systems is to offer DSM-like behavior but with emphasis on automation and not performance. The partitioned applications run on existing infrastructure (e.g., DCOM or regular JVMs) but relieve the programmer of the burden of dealing with the idiosyncrasies of various middleware mechanisms. However, this reduces the field of application to programs whose locality patterns are very well-known—otherwise performance could be affected significantly. Systems such as Addistant [TSCI01], J-Orchestra [TS02], and Pangaea [Spi02] are examples of automatic partitioning tools in the Java world.

JavaParty System. The JavaParty system [HRP, PZ97] 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 our approach, JavaParty is much closer to a DSM system, as it incurs similar overheads and employs similar mechanisms for exploiting locality.

Finally, we should mention that approaches that hide the fact that a network is present have often been criticized [KWWW94]. The main point of this 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. Our approach follows RMI's design principle of making partial failure apparent to the programmer. Thus, remote methods in a declarative approach can throw remote exceptions that the programmer is responsible for handling.

Chapter 7

Future Work and Conclusions

7.1 Future Work

A promising future work direction is to develop an annotation-based distributed object system for an emerging object-oriented language, such as *Ruby* [TH01]. It would be interesting to explore how advanced language features such as built-in aspects, closures, and co-routines can be utilized in the implementation. Despite its exploratory nature and the presence of advanced features, Ruby's distributed object system, *DRuby* [Sek07], does not significantly differ from Java RMI. An empirical study could then reveal how an annotation-based approach to remote parameter passing affects software engineering practices.

A modern language such as *Scala* [OAC+04] has built-in support for language extensibility. The very name stems from the language's scaling capabilities. Scala integrates object-

oriented and functional programming concepts in a statically typed language. The two programming styles blend well contributing towards Scala's extensibility. While its functional programming constructs enable greater expressiveness, its object-oriented constructs ease the task of structuring large systems. Together, they facilitate the expression of new kinds of programming patterns and component abstractions, with a concise programming style. Some of the language extensibility features provided by Scala include, extensible types, extensible control structures, operator overloading etc. These features provide fine-grained control over the language capabilities. In addition, the Scala programs interoperate seamlessly with Java and can make use of Java APIs. We would like to explore how these features aid the development of a declarative parameter passing model for Scala.

Being a part of the remote method signature, the remote parameter passing annotations must appear in both the method declaration in the remote interface and the method definitions in all remote classes implementing the interface. To relieve the programmer of this burden, we could develop a plugin for a modern IDE such as Eclipse [Fou07], which would generate the declarative style method signatures as a part of the generated stub implementations for the remote interface.

Finally, our framework uses AspectJ [KHH+01] for introducing cross-cutting concerns. A possible future work is to use a distributed, dynamic AOP technique such as Java Aspect Components (JAC) [PSD+04], which enables advices to be inserted or removed on the fly instead of AspectJ.

7.2 Conclusions

This thesis has provided a clear exposition of the shortcomings of a type-based remote parameter passing model. To overcome these shortcomings, we presented an annotation-based parameter passing approach in distributed object systems as a better alternative to a type-based parameter passing approach. We also provided an argument in favor of treating parameter passing in distributed object systems as a separate concern. Based on this principle, we presented an extensible framework for annotation-based parameter passing and described how multiple different semantics can be efficiently implemented on top of a type-based parameter passing model with ease using our extensible framework.

We believe that our framework provides a powerful distributed programming platform and a convenient experimentation facility for research in distributed object systems.

Bibliography

[AFT99] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *International Conference on Parallel Processing*, volume 411, 1999.

[ALS06] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. 2006.

[BBH+98] H.E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, Tim Ruhl, and M.F. Kaashoek. Performance Evaluation of the Orca Shared-Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, 1998.

[BHJL07] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The development of the emerald programming language. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 11–1–11–51, 2007.

[BK98] N. Brown and C. Kindel. Distributed Component Object Model Protocol–DCOM/1.0, 1998. Redmond, WA, 1996.

[BN84] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[CBZ91] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 152–164, 1991.

[Cor07] Microsoft Corporation. Visual Studio 2005, 2007. `msdn.microsoft.com/vstudio`.

[Dah00] Markus Dahm. Doorasthaa step towards distribution transparency. In *Proceedings of the Net. Object Days 2000*, 2000.

[DLFM96] A. De Lucia, A.R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In 4th *IEEE Workshop on Program Comprehension*, pages 9–18, 1996.

[EFF+97] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–56, 1997.

[EM04] M. Eichberg and M. Mezini. Alice: Modularization of Middleware using Aspect-Oriented Programming. *Software Engineering and Middleware (SEM)*, 2004, 2004.

[ET01] J. Eberhard and A. Tripathi. Efficient Object Caching for Distributed Java RMI Applications. *Lecture Notes In Computer Science*, 2218:15–35, 2001.

[Eug03] Patrick Th. Eugster. Lazy Parameter Passing. Technical report, 2003.

[Eug06] Patrick Eugster. Uniform proxies for java. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 139–152, New York, NY, USA, 2006. ACM Press.

[FBF03] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pages 399–413, 2003.

[FHL94] B. Ford, M. Hibler, and J. Lepreau. Separating presentation from interface in RPC and IDLs. 1994.

[FHL95] B. Ford, M. Hibler, and J. Lepreau. Using annotated interface definitions to optimize RPC. *ACM SIGOPS Operating Systems Review*, 29(5), 1995.

[Fou07] The Eclipse Foundation. Eclipse - an open development platform, 2007. `http://www.eclipse.org`.

[FR03] M. Fleury and F. Reverbel. The JBoss Extensible Server. *International Middleware Conference*, 2003.

[Gro98a] Object Management Group. Objects By Value. document orbos/98-01-18, 1998. Framingham, MA.

[Gro98b] Object Management Group. The Common Object Request Broker: Architecture and Specification, 1998. Framingham, MA.

[Gro03] Object Management Group. IDL to Java Language Mapping Specification, 2003. Framingham, MA.

[HOT02] W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. *IBM Research Report RC22685 (W0212-147) December*, 30, 2002.

[HRP] B. Haumacher, J. Reuter, and M. Philippsen. JavaParty: A distributed companion to Java. `http://wwwipd.ira.uka.de/JavaParty/`.

[Inc02] Cincom Systems Inc. Opentalk Communication Layer Developer's Guide, 2002. `http://www.cincom.com/downloads/pdf/OpentalkDevGuide.pdf`.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072(327-355):110, 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *ECOOP'97–object-oriented Pro-*

gramming: 11th European Conference, Jyväskylä, Finland, June 9-13, 1997: Proceedings, 1997.

[KWWW94] S.C. Kendall, J. Waldo, A. Wollrath, and G. Wyant. A Note on Distributed Computing. 1994.

[Lop96] C.V. Lopes. Adaptive Parameter Passing. In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS96)*. Springer-Verlag, 1996.

[Mic07] Sun Microsystems. NetBeans IDE, 2007. `http://www.netbeans.org`.

[MvNV+99] J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 173–182. ACM Press New York, NY, USA, 1999.

[MVNV+01] J. Maassen, R. Van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, 2001.

[NCT04] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed AOP. *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15, 2004.

[OAC+04] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. *LAMP-EPFL*, 2004.

[OH01] Piet Obermeyer and Jonathan Hawkins. Microsoft .NET Remoting: A Technical Overview. *MSDN Library*, July 2001.

[orb07] Object Request Broker, 2007. http://www.sei.cmu.edu/str/descriptions/orb.html.

[Paw05] R. Pawlak. Spoon: annotation-driven program transformation—the AOP case. *Proceedings of the 1st workshop on Aspect oriented middleware development*, 2005.

[PHN00] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency Practice and Experience*, 12(7):495–518, 2000.

[Pre97] C. Prehofer. Feature-Oriented Programming: AFresh Look at Objects. *ECOOP'97–object-oriented Programming: 11th European Conference, Jyväskylä, Finland, June 9-13, 1997: Proceedings*, 1997.

[PSD+04] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC: an aspect-based distributed dynamic framework. *Software Practice and Experience*, 34(12):1119–1148, 2004.

[PZ97] M. Philippsen and M. Zenger. JavaParty– transparent remote objects in Java. *Concurrency Practice and Experience*, 9(11):1225–1242, 1997.

[RWWB96] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling State in the Java System. *Computing Systems*, 9(4):291–312, 1996.

[SB98] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570, 1998.

[Sek07] Masatoshi Seki. DRuby–A Distributed Object System for Ruby, 2007. `http://www.ruby-doc.org/stdlib/libdoc/drb/`.

[SM99] T.S. Souder and S. Mancoridis. A Tool for Securely Integrating Legacy Systems into a Distributed Environment. *Working Conference on Reverse Engineering*, pages 47–55, 1999.

[SM02] M. Surdeanu and D. Moldovan. Design and performance analysis of a distributed Java Virtual Machine. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):611–627, 2002.

[Spi02] A. Spiegel. *Automatic Distribution of Object Oriented Programs.* PhD thesis, Berlin, Freie University, 2002.

[SRSS00] D.C. Schmidt, H. Rohnert, M. Stal, and D. Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* John Wiley & Sons, Inc. New York, NY, USA, 2000.

[Sun97a] Sun Microsystems. *Java Remote Method Invocation Specification*, 1997.

[Sun97b] Sun Microsystems, February. *Java Object Serialization Specification*, 1997.

[TH01] D. Thomas and A. Hunt. *Programming Ruby.* Addison-Wesley Reading, MA, 2001.

[TS02] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2002.

[TS08] E. Tilevich and Y. Smaragdakis. NRMI: Natural and Efficient Middleware. *IEEE Transactions on Parallel and Distributed Systems*, pages 174–187, February 2008.

[TSCI01] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of Legacy Java Software. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2001.

[TTK98] G.K. Thiruvathukal, L.S. Thomas, and A.T. Korczynski. Reflective remote method invocation. *Concurrency - Practice and Experience*, 10(11-13):911–925, 1998.

[WJD03] E. Wohlstadter, S. Jackson, and P. Devanbu. DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *Proceedings of the International Conference on Software Engineering*, volume 186, 2003.

[YC97] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency - Practice and Experience*, 9(11):1213–1224, 1997.

[YCC+06] C.C. Yang, C.K. Chen, Y.H. Chang, K.H. Chung, and J.K. Lee. Streaming support for Java RMI in distributed environments. *Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 53–61, 2006.

[ZJ03] C. Zhang and H. Jacobsen. Refactoring middleware with aspects. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1058–1073, 2003.

Appendix A

Performance

We conducted a series of micro-benchmarks comparing the performance of pass by *copy* and pass by *remote-reference* semantics implemented as DeXteR plugins with that of the native implementation. The results represent the average of running each benchmark 1,000 times on a Pentium D 3.GHz (dual core) machine with 2GB of RAM, running Sun JVM version 1.6.0. By warming the JVM, we ensured that the measured programs had been dynamically compiled before measurements.

Since pass by *copy* predominantly involves the cost of object serialization, its micro-benchmark involved measuring the execution times for a varying object size. Figure A.1 presents the performance comparison of the two implementations of pass by *copy*.

In lieu of support for type-compatible dynamic proxies for classes in Java, our implementation emulates this functionality using a type-incompatible client-side dynamic proxy and a type-

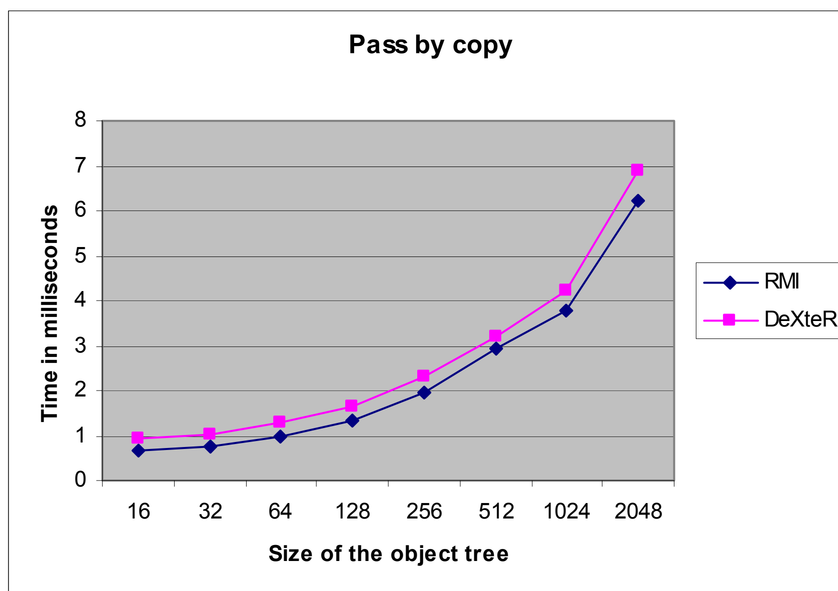

Figure A.1: Pass by copy benchmark.

compatible server-side wrapper proxy. Thus, this emulated functionality introduces two new levels of indirection compared to the standard Java RMI implementation of pass by remote-reference. As any new level indirection inherently introduces some performance overhead, it is important to verify that this overhead is not prohibitively expensive. The purpose of passing a parameter by remote-reference is to enable the server to invoke methods on that parameter as part of the logic of the remote method. Since these invocations will be propagated back to the client, these method invocations are called remote callbacks. Figure A.2 presents the performance comparison between the two implementations of pass by *remote-reference*, for a varying number of remote callbacks.

As the latency of a remote call is orders of magnitude greater than that of a local call, we expect that the overhead incurred by the additional local calls added to the remote call by

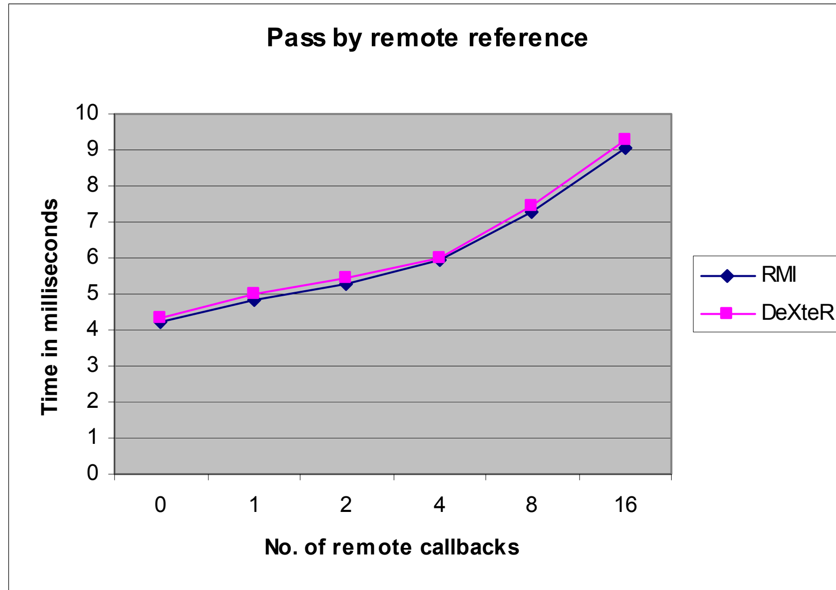

Figure A.2: Pass by remote-reference benchmark.

DeXteR is negligible. Our initial experiments confirm just that, showing that our approach is feasible.