SOFTWARE TOOLS FOR SEPARATING DISTRIBUTION CONCERNS

A Thesis
Presented To
The Academic Faculty

by

Eli Tilevich

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
December 2005

SOFTWARE TOOLS FOR SEPARATING DISTRIBUTION CONCERNS

Approved by:

Dr. Yannis Smaragdakis, Advisor
College of Computing
Georgia Institute of Technology

Dr. Mary Jean Harrold
College of Computing
Georgia Institute of Technology

Dr. Doug Lea
Computer Science Department
State University of New York at Oswego

Dr. Santosh Pande
College of Computing
Georgia Institute of Technology

Dr. Karsten Schwan
Computer Science Department
Georgia Institute of Technology

Date Approved: October 31, 2005

To my family,

for their love and support.

ACKNOWLEDGEMENTS

It's a sign of mediocrity when you demonstrate gratitude with moderation.
Roberto Benigni

My graduate school experience was about intensive learning not only in the field of computer science but also in many other aspects of life, and now I'm a different (hopefully wiser and more mature) person as a result of this experience. As is the case with any large endeavor, many people have contributed to the success of this dissertation.

First of all, I would like to thank my family for their support, encouragement, and unshakable faith in my abilities during the course of my studies. Without their support, inspired by tradition and the belief of the European intelligentsia that nothing is more valuable and important than education, I may not have persevered in my studies.

Regular telephone conversations with my mother saved me a fortune in psychotherapy costs. Whenever I felt dejected or doubtful due to diverse factors such as making slower than expected progress in my research, learning that one of my papers was rejected, having no personal or social life whatsoever, or imagining the life I could have had, had I chosen to continue my professional career in the "real world," my mother would assure me that pursuing a Ph.D. was the right thing to do and that the pleasures of normal life could wait. My family has exhibited such incomprehensible pride about my pursuing a doctorate degree that I have always felt like I had no choice but to carry on and successfully complete my studies.

I would like to thank my advisor Yannis Smaragdakis and all the members of my committee: Mary Jean Harrold, Doug Lea, Santosh Pande, and Karsten Schwan.

Meeting my advisor Yannis Smaragdakis solidified my intent to pursue academic research training. He gave me the opportunity to work on problems that I found exciting and gently guided me through the challenges of mastering the intricacies of the research trade. His explanation that the only worthy reason for getting a Ph.D. is to become a "fully trained Jedi Knight" seemed just barely convincing five years ago, but it makes perfect sense to me now. His knowing exactly when a hands-on or hands-off management approach was most appropriate helped me immensely in my development as a researcher. His pointing out the differences between analytic and synthetic abilities and his advice that I work on improving the former helped me focus my efforts and achieve maximum benefits. Yannis has never failed to amaze me with the depth of his analysis and insight, and I can only hope that I'd be as effective an advisor for my future students as Yannis was for me.

I was fortunate to have an office on the same floor as that of Mary Jean Harrold, whom I worked for as an RA for one semester. I quickly learned that the only way to get to work earlier than Mary Jean was to stay in my office all night. I have learned so much by observing Mary Jean interact with her students, and I can only hope that some of her incredible work ethics, dedication to her students, and passion for academic work have rubbed off on me.

I was also fortunate to have Doug Lea as the external member of my committee. Every interaction with Doug was a learning experience of such high quality that I felt compelled to take notes whenever I was talking to him. I find Doug's dedication to science and work ethics somewhat intimidating—I could not imagine that a researcher of his stature and with such a busy schedule would not only read all my dissertation documents but also provide multiple insightful comments and suggestions.

The support and encouragement that I received from Santosh Pande have greatly contributed to the success of this dissertation. His advice to “hold my ground” proved to be invaluable. In addition, he contributed interesting ideas for future work directions of this dissertation.

Karsten Schwan firmly declared, after he had known me for just a few weeks, that I appeared to be “a systems person” and that I should pursue research in systems. His uncanny intuition about the area for which I was most suited at a time when I was oblivious to my proclivity or talents in that particular research area helped guide my studies. His guidance was invaluable, but more importantly, his encouragement and appreciation of my work greatly boosted my confidence and helped me succeed throughout my experience at Tech.

The exchange Masters students from the University of Stuttgart, Marcus Handte and Stephan Urbanski, greatly helped me in creating the infrastructure for my research. Marcus built the J-Orchestra GUI and contributed to the code base, and Stephan implemented most of the GOTECH framework. I also would like to acknowledge the help of other Masters students: Austin Chau, Nikitas Liogkas, and Dean Mao. Nikitas was a major contributor to the Kimura case study.

I’d like to thank all the members of the INCITE Lab: Idris Hsi, Vernard Martin, Jochen “Je77” Rick, Patrick Yaner, Dave Zook, and others, for engendering an atmosphere of intellectual vitality, exchange, and collegiality in our lab. Listening to and participating in their discussions on various matters provided me with an opportunity to engage in intellectual discourse beyond my disciplinary boundaries. Thanks to them, I’m a more well-rounded thinker than I used to be.

I'd like to thank all the present and former members of Mary Jean Harrold's Aristotle research group: Taweesup "Term" Apiwattanapong, George Baah, Jim Bowring, Pavan Kumar Chittimalli, Jim Jones, Donglin Liang, Maikel Pennings, Saurabh Sinha, and others. I can't imagine a better bunch of people as grad school colleagues! Being able to walk across the hall at any time and engage them in meaningless conversation while preventing them from doing their work helped me maintain my sanity. Also, competing with and invariably losing to Term in trying to lift a bigger percentage of one's weight during our regular workouts did not just keep me in shape but also helped me tremendously in dealing with the stress of graduate school.

I'd like to thank all the SPARC faculty: Pete Manolios, Alex Orso, Spencer Rugaber, and Olin Shivers for generously sharing their wisdom with me, as well as the SPARC students: Christoph Csallner, David Fisher, Shan Shan Huang, Lex Spoon, and others for being a receptive and critical audience on which I could try out my research ideas.

I'd like to thank Deborah Mitchell, Barbara Binder, and Jennifer Chisholm for making all the administrative matters a cinch to deal with.

Because of a freak accident that put an end to my career as a professional musician, my experience with computers started quite haphazardly. This experience would not have been as successful without the fine instruction, helpful advice, and strong encouragement I received at Pace University in New York. Among the many fine professors that I had the pleasure of interacting with as a student at Pace, Carol Wolf and Joseph Bergin are the two to whom I'm particularly grateful. Dr. Wolf was my first professor of programming, and because of her exceptional teaching, I have not only mastered the fundamentals of computer science but also gained confidence in my abilities as a computer programmer and in

my future success. By distinguishing me as a student with strong potential for becoming a good researcher, Dr. Bergin gave me the confidence necessary to pursue a doctorate degree. Furthermore, his personal involvement in the application for admission process was instrumental in my admission to Georgia Tech.

Finally, I'd like to thank the faculty at Virginia Tech for giving me a job!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

With the advent of the Internet, distributed programming has become a necessity for the majority of application domains. Nevertheless, programming distributed systems remains a delicate and complex task. This dissertation explores separating distribution concerns, the process of transforming a centralized monolithic program into a distributed one. This research develops algorithms, techniques, and tools for separating distribution concerns and evaluates the applicability of the developed artifacts by identifying the distribution concerns that they separate and the common architectural characteristics of the centralized programs that they transform successfully. The thesis of this research is that software tools working with standard mainstream languages, systems software, and virtual machines can effectively and efficiently separate distribution concerns from application logic for object-oriented programs that use multiple distinct sets of resources. Among the specific technical contributions of this dissertation are (1) a general algorithm for call-by-copy-restore semantics in remote procedure calls for linked data structures, (2) an analysis heuristic that determines which application objects get passed to which parts of native (i.e., platform-specific) code in the language runtime system for platform-independent binary code applications, (3) a technique for injecting code in such applications that will convert objects to the right representation so that they can be accessed correctly inside both application and native code, (4) an approach to maintaining the Java centralized concurrency and synchronization semantics over remote procedure calls efficiently, and (5) an approach to enabling the execution of legacy Java code remotely from a web browser.

The technical contributions of this dissertation have been realized in three software tools for separating distribution concerns: NRMI, middleware with copy-restore semantics; GOTECH, a program generator for distribution; and J-Orchestra, an automatic partitioning system. This dissertation presents several case studies of successfully applying the developed tools to third-party programs.

CHAPTER I

INTRODUCTION

As the emergence of the Internet has changed the computing landscape, distribution has become a necessity in a large and growing number of software systems. The focus of distributed computing has been shifting from "distribution for parallelism" to "resource-driven distribution," in which the resources of an application are naturally remote from each other or from the computation. Because of this shift, more and more centralized applications, written without any distribution in mind, are being adapted for distributed execution. This entails adding distributed capabilities to these applications to move parts of their execution functionality to a remote machine. Examples abound: a local database grows too large and is relocated to a powerful server, becoming remote from the rest of the application; a desktop application needs to redirect its output to a remote graphical display or to receive input from a remote digital camera or a sensor; a desktop application, executed on a PDA, does not find all the hardware and software resources that it references available locally and needs to access them remotely; or a software component, designed for local access, is distributed over a network and needs to be accessed remotely.

These examples introduce the issue of separating distribution concerns. Separation of concerns has been a guiding principle for controlling the complexity of software ever since Dijkstra [20] coined the term almost 30 years ago. As described by Ossher and Tarr [64], separation of concerns is "the ability to identify, encapsulate and manipulate only

those parts of software that are relevant to a particular concept, goal, or purpose." The advent of the Java technology re-ignited interest in the subject within the software research community, with industry not far behind, resulting in such tools as AspectJ [41] and HyperJ [28]. In light of these developments, the question of which concerns can be effectively and efficiently separated has taken on a new significance and importance.

As many other principles of computing, separating computational concerns encompasses two dimensions: what and how. While the "what" dimension refers to determining whether a particular concern can be separated and identifying the specific code entities expressing it, the "how" dimension pertains to how actual separation can be realized at the implementation level.

Some concerns fundamentally define the meaning of computation and as such cannot be separated. For instance, parallel algorithms often have no resemblance to sequential algorithms for the same problem, and some problems are very unlikely to even have an efficient parallel solution. Thus "efficient parallelism" is not a concern that can be separated from the logic of a software application. (Similar arguments apply to transactions and failure handling as well [43].)

In view of such difficulties, most research has shifted from the problem of separating concerns to the problem of removing low-level technical barriers to the separation of concerns, assuming that the separation is conceptually possible. In software tools, two main directions have been identified. The first is that of general-purpose tools for expressing different concerns as distinct code entities and composing them together. The second is that of domain-specific tools that achieve separation of concerns for well-defined domains by hiding such concerns behind programming abstractions (e.g., new language constructs).

The term "aspect-oriented" is often used to describe the first direction (although it was originally [40] proposed as a concept that encompasses both directions).

While many domains can derive benefits from separating concerns, in recent years, it is the area of enterprise computing in which such benefits have become particularly evident. Through the process, which is currently being standardized [37], some J2EE application servers use the so-called aspect-oriented programming (AOP) frameworks to add various non-functional capabilities such as caching, security, and persistence to enterprise business objects.

In the context of this dissertation, the term "separation of distribution concerns" refers to the process of transforming a centralized monolithic program into a distributed one. In other words, we treat distribution functionality as a separate concern that is added to the application logic of a centralized program, thereby transforming it into a distributed program. It has long been under debate whether distribution is a concern that can be at all separated from application logic. For example, Waldo et al.'s well-known "A Note on Distributed Computing" [96] argues that "papering over the network" is ill-advised. The main reasons include difference in performance, different calling semantics, and the possibility of partial failure. (Other reasons, mentioned by Waldo et al., such as direct memory access, no longer hold today for Java and C#.) The research described in this dissertation does not attempt to refute any of the major arguments made by the authors of the Note: our answer to the question of whether distribution can be successfully introduced transparently to *all* programs is still a resounding no.

To clarify our perspective, let us consider the two extremes that define the transparency spectrum of approaches to separating distribution concerns: "papering over the net-

work" and the so-called "explicit" approach. At one extreme, "papering over the network" is a completely transparent approach to distribution that masks all the differences between the centralized and distributed execution models from the programmer. Some distributed shared memory (DSM) systems follow this approach. At the other extreme, the "explicit" approach makes use of different programming idioms for distributed computing as a means to accommodate for the differences in performance, calling semantics, and the possibility of partial failure. A representative of this approach is Java RMI [80] itself. Taking the middle ground between these two extremes from the transparency perspective, this research follows the approach to separating distribution concerns whose unifying theme can be defined as "translucency." Our approach is "translucent" in the sense that it is trying to be transparent, but without going all the way. In other words, our approach aims at creating software tools for distributed computing that are more convenient to use from the programmer's perspective (i.e., closer to the familiar centralized programming model), while being fully-aware of the differences between the centralized and distributed execution models.

This research delineates the limits of introducing distribution translucently through the following three steps. First, we determine which distribution concerns, defined as the differences between centralized and distributed execution, can be separated effectively and efficiently. Second, we outline the architectural characteristics of a class of programs to which distribution can indeed be introduced translucently. Third, in trying to achieve distribution translucency, we make improvements to several mainstream software tools for distributed computing such as RPC middleware [10]. Because adding distributed capabilities to existing programs is currently one of the most important software evolution tasks in

practice [44], the improved software tools for separating distribution concerns are valuable even if successful distribution requires changes to the application logic.

The primary goal of this research is to explore novel software tools for separating distribution concerns that, for a certain class of object oriented programs, bridge centralized and distributed programing models and semantics as closely as possible. Taking the software tools approach to this problem entails that in transforming a centralized monolithic program into a distributed one only the program's code itself changes, while the runtime system remains intact. That is, the new software tools, explored by this research, work exclusively with standard mainstream languages, systems software, and virtual machines.

1.1 Overview of Software Tools

NRMI [88], middleware with copy-restore semantics, GOTECH [89], a program generator for distribution, and J-Orchestra [87], an automatic partitioning system are three developed software tools for evolving a centralized program into a distributed one. Although these tools overlap in terms of the kinds of distribution concerns that they separate, each one addresses the general problem from a different perspective, makes different assumptions about the original centralized programs to which it can be applied, and introduces novel algorithms, techniques, and tools applicable to different programming scenarios.

1.1.1 Middleware with Copy-Restore Semantics

The NRMI middleware system [88] supports call-by-copy-restore semantics in addition to traditional call-by-copy semantics. Intuitively, this means that NRMI allows the user to specify that changes to data reachable by the arguments of a remote method be repro-

duced on the caller site. In addition, NRMI does this in full generality, even for complex, pointer-based data structures, imposing very low computation and communication overheads (remote calls proceed at full speed). Call-by-copy-restore semantics is highly desirable in distributed computing because it causes remote calls to behave exactly like local calls in many cases (e.g., in the important case of single-threaded clients and stateless servers). Both the value of call-by-copy-restore and the need for a mechanism to support it in full generality has been repeatedly identified in the distributed systems community. In their recent textbook Distributed Systems (2002), Tanenbaum and van Steen summarize the problem that NRMI was the first middleware to solve:

> *Although [call-by-copy-restore] is not always identical [to local execution], it frequently is good enough. ...[Current call-by-copy-restore mechanism] still cannot handle the most general case of a pointer to an arbitrary data structure such as a complex graph.*

1.1.2 Program Generation for Distribution

Sometimes the problems of programming distributed systems are purely those of conciseness and expressiveness of the language tools. In this direction, we have developed the GOTECH program generator [89], which accepts programmer-supplied annotations and generates distribution code, relieving the programmer from writing tedious, protocol-specific code by hand. GOTECH depends only on general-purpose tools, offers easy-to-evolve implementation amenable to inspection and change, and uniquely combines aspect-oriented and generative techniques. In general, domain-specific tools that automate rote programming tasks are of significant interest from a software design standpoint.

1.1.3 Automatic Partitioning

The process of rewriting a centralized application using a compiler-level tool in order to produce its distributed version is called automatic partitioning. This approach is more automated than copy-restore middleware and program generation for distribution. Although the process cannot be fully automated, most correctness aspects of the rewrite are typically handled automatically (i.e., the resulting distributed application has semantics identical to the original centralized one) while performance aspects are optimized under user guidance. Automatic partitioning is a relatively new approach: only a handful of partitioning tools exist, and all of them have been developed in the past five years. Nevertheless, the goal of automatic partitioning is almost identical to that of distributed shared memory (DSM) systems, a mature systems area. The difference is in the techniques used: DSMs operate by providing a system (i.e., a runtime environment) that enables distributed execution. In contrast, automatic partitioning tools take a language approach and rewrite the application only without making any change to the runtime environment. The difference has a significant practical implication: an automatically partitioned application can be deployed very easily in standard runtime environments without any need for specialized support. For example, a partitioned Java application can run on any Java-enabled platform, from PDAs and cell phones to mainframes.

We have developed the J-Orchestra automatic partitioning system for Java programs [87]. J-Orchestra, arguably the most mature and scalable automatic partitioning system in existence, was the first system to identify the presence of unmodifiable code in the runtime system that can access regular language-level objects (e.g., Java VM code for opening file objects) as a salient problem with automatic partitioning. If such code accesses a remote

object, a runtime error will occur since the code is unaware of distribution (e.g., it expects to access fields of a regular object but instead receives a proxy). J-Orchestra addresses this problem with a rewrite algorithm that automatically transforms object references from direct to indirect at run-time, ensuring that they are in the correct form for the code that handles them. As a result, J-Orchestra has scaled to realistic, third-party applications. Also, the ease of creating distributed programs with J-Orchestra as compared to programming with standard distribution middleware has demonstrated automatic partitioning as a promising technology for prototyping ubiquitous computing applications [51].

1.2 Thesis Statement

> *Software tools working with standard mainstream languages, systems software, and virtual machines can effectively and efficiently separate distribution concerns from application logic for object-oriented programs that use multiple distinct sets of resources.*

This research proves this thesis through a two-phase process. The first phase develops algorithms, techniques, and tools for separating distribution concerns. We will refer to the deliverables of this phase of research as "research artifacts." The second phase evaluates the applicability of the developed research artifacts in terms of their effectiveness and efficiency. We will evaluate these artifacts by determining (1) the exact set of distribution concerns that they separate and outlining (2) the common architectural characteristics of the centralized applications that they can transform effectively and efficiently.

1.3 Contributions

The contributions of this research include:

1. a general algorithm for call-by-copy-restore semantics in remote procedure calls for linked data structures,

2. an analysis heuristic that determines which application objects get passed to which parts of native (i.e., platform-specific) code in the language runtime system for platform-independent binary code applications,

3. a technique for injecting code in such applications that will convert objects to the right representation so that they can be accessed correctly inside both application and native code,

4. an approach to maintaining the Java centralized concurrency and synchronization semantics over remote procedure calls efficiently, and

5. an approach to enabling the execution of legacy Java code remotely from a web browser.

1.4 Overview of Distribution Concerns

Table 1-1. Distribution Concerns and Solutions

Challenges of Separating Distribution Concerns	Solutions
Semantics	
• The lack of shared address space; the difference in parameter-passing semantics. • Distribution in the presence of unmodifiable (system: OS, JVM) code.	• NRMI and its efficient implementation of the call-by-copy-restore semantics. • The J-Orchestra approach to enabling indirection even in the presence of unmodifiable code (analysis heuristics, a novel rewrite algorithm, and run-time direct-indirect and vice verse translation).
Performance	
• The latency of a remote call is several orders of magnitude slower than that of a local one.	• J-Orchestra: • profiling, • object mobility, and • placement policy based on creation site.
Distribution Middleware Conventions	
• Having to deal with the complex conventions of using modern middleware mechanisms. • Preserving the centralized concurrency and synchronization semantics in a distributed environment.	• The NRMI call-by-copy-restore is more natural than the standard call-by-copy. • Combining generative and aspect-oriented techniques in a novel way in GOTECH to automate the complexities of enabling server-side distribution in J2EE. • The J-Orchestra approach to dealing with distributed multi-threading and synchronization.
Viability & Scalability	
	• Various case studies.

1.5 Overview of Dissertation

The rest of this dissertation is structured as follows. The chapters II, III, and IV cover the motivation, design, and implementation of NRMI, GOTECH, and J-Orchestra, respectively. Chapter V discusses various applicability issues and validation through case studies. Chapter VI demonstrates how the J-Orchestra indirection machinery can be extended to domains other than distributed computing. Chapter VII presents related work. Chapter VIII concludes after discussing future research directions and the merits of this dissertation.

CHAPTER II

NRMI

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

2.1 Introduction

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

serious restriction and one that has often been identified. The recent (2002) Tannenbaum and van Steen "Distributed Systems" textbook [83] summarizes the problem and (most) past approaches:

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

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

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

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

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

In summary, this chapter presents the following insights:

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

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

2.2 Background and Motivation

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

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

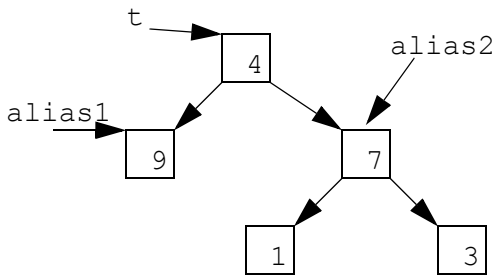


Figure 2-1: A tree data structure and two aliasing references to its internal nodes.

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

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

```
void foo(Tree tree) {

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

}
```

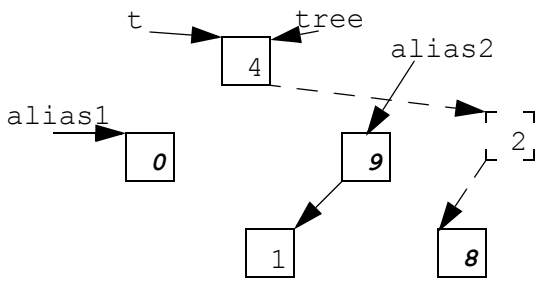


Figure 2-2: A local call can affect all reachable data

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

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

Consider now what happens when `foo` is a remote method, implemented by a server on a different machine. An obvious solution would be to maintain call-by-reference seman-

tics by introducing “remote references” that can point to data in a different address space, as shown in Figure 2-3.

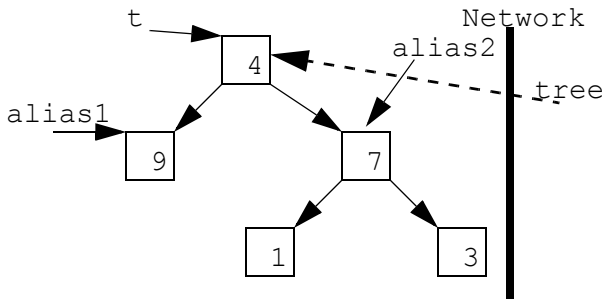


Figure 2-3: Call-by-reference semantics can be maintained with remote references.

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

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

Nevertheless, the usual semantics for reference data in RMI calls (and the vast majority of other middleware) is *call-by-copy*. (“Call-by-copy” is really the name used in the Distributed Systems community for *call-by-value*, when the values are complex data structures.) When a reference parameter is passed as an argument to a remote routine, all data reachable from the reference are deep-copied to the server side. The server then operates on the copy. Any changes made to the deep copy of the argument-reachable data are

not propagated back to the client, unless the user explicitly arranges to do so (e.g., by passing the data back as part of the return value).

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

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

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

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

- call-by-copy-restore has to "overwrite" the original data objects (e.g., `t.right.data` in our example), not just link new objects in the structure reachable from the reference argument of the remote call (`t` in our example). The reason is that, at the client site, the

objects may be reachable through other references (`alias2` in our example) and the changes should be visible to them as well.

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

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

2.3 Supporting Copy-Restore

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

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

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

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

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

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

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

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

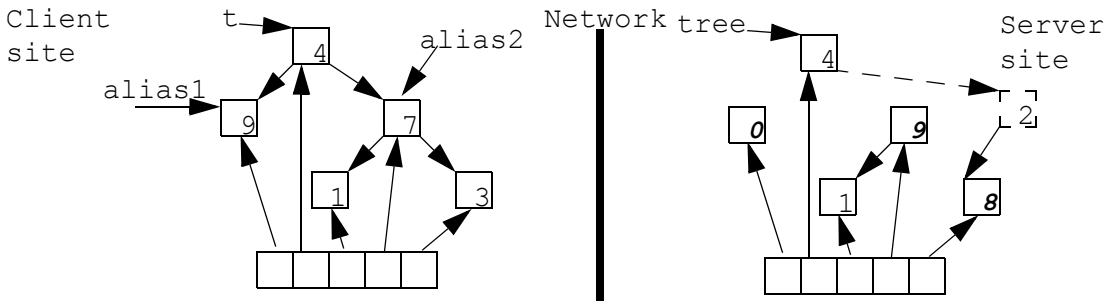


Figure 2-4: State after steps 1 and 2 of the algorithm. Remote procedure `foo` has performed modifications to the server version of the data.

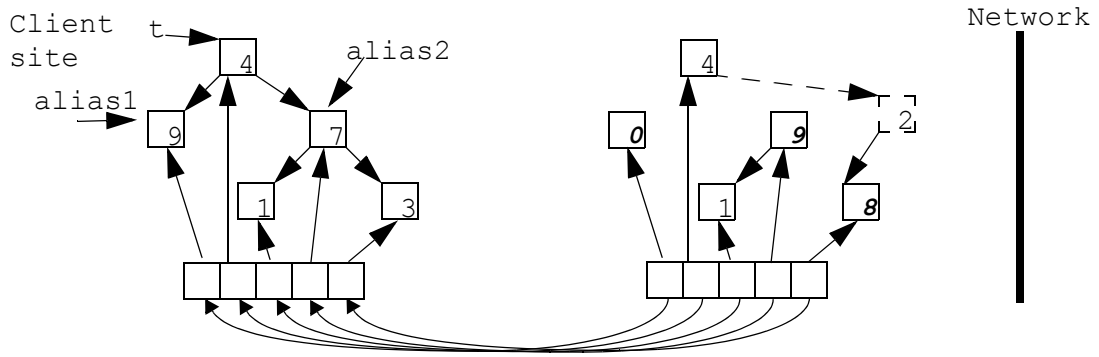


Figure 2-5: State after steps 3 and 4 of the algorithm. The modified objects (even the ones no longer reachable through `tree`) are copied back to the client. The two linear representations are "matched"—i.e., used to create a map from modified to original versions of old objects.

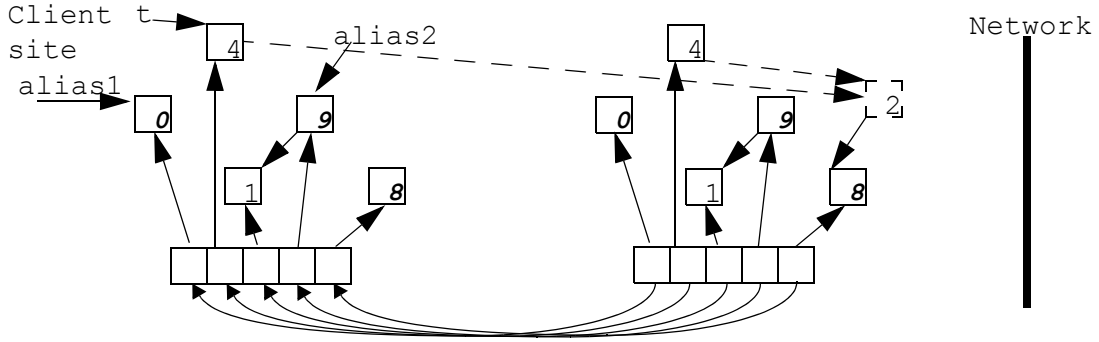


Figure 2-6: State after step 5 of the algorithm. All original versions of old objects are updated to reflect the modified versions.

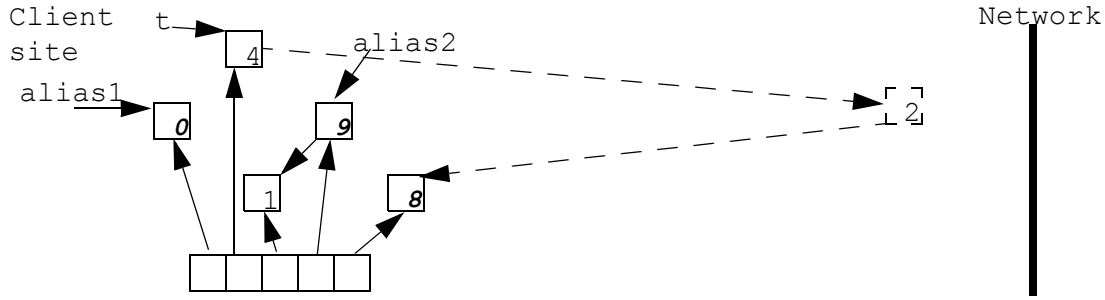


Figure 2-7: State after step 6 of the algorithm. All new objects are updated to point to the original versions of old objects instead of their modified versions. All modified old objects and their linear representation can now be deallocated. The result is identical to Figure 2-3.

2.4 Discussion

2.4.1 Copy-Restore vs. Call-by-Reference

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

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

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

2.4.2 DCE RPC

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

DCE RPC supports three different kinds of pointers, only one of which (*full pointers*) supports aliasing. DCE RPC full pointers, declared with the `ptr` attribute, can be safely aliased and changed by the callee of a remote call. The changes will be visible to the caller, even through aliases to existing structure. Nevertheless, DCE RPC only guarantees

correct updates of aliased data for aliases that are declared in the parameter lists of a remote call.[1] In other words, for pointers that are not reachable from the parameters of a remote call, there is no guarantee of correct update.

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

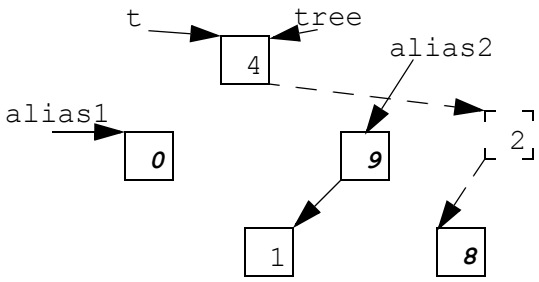


Figure 2-8: Changes after execution of method.

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

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

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

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

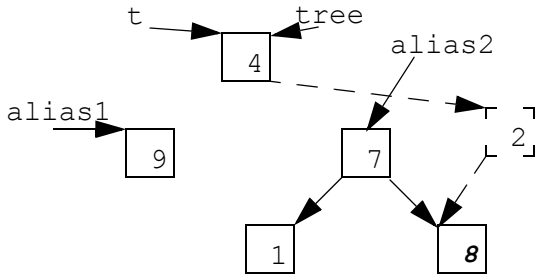


Figure 2-9: Under DCE RPC, the changes to data that became unreachable from `t` will not be restored on the client site.

2.5 NRMI Implementations

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

NRMI currently has three implementations, each applicable to different programming environments and scenarios. The implementation in the form of a full, drop-in replacement for Java RMI demonstrates how this standard middleware mechanism for the Java language can be transparently enhanced with call-by-copy-restore capacities. However, introducing a new feature into the implementation of a standard library of a mainstream programming language is a significant undertaking, requiring multiple stakeholders in the Java technology to reach a consensus. Therefore, our other two implementations pro-

vide Java programmers with call-by-copy-restore capacities without having to change any of the standard Java libraries. One implementation takes advantage of the extensible application server architecture offered by JBoss [68] to introduce NRMI as a pair of client/server interceptors. Another introduces NRMI by retrofitting the bytecodes of application classes that use the standard RMI API. Having to work around the inability to change the RMI runtime libraries, these latter two solutions are not always as efficient as the drop-in replacement one but offer interesting insights on how new middleware features can be introduced transparently. Therefore, we limit our discussion on various optimization issues of NRMI to the RMI drop-in replacement implementation only.

2.5.1 A Drop-in Replacement of Java RMI

2.5.1.1 Programming Interface

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

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

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

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

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

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

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

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

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

Declaring a class to implement `java.rmi.Restorable` is all that is required from the programmer: NRMI will pass all instances of such classes by-copy-restore whenever

they are used in remote method calls. The NRMI runtime handles the restore phase of the algorithm totally transparently to the programmer. This saves lines of tedious and error-prone code as we illustrate in Section 5.2.

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

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

2.5.1.2 Implementation Insights

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

Creating a linear map

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

Performing remote calls

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

Updating original objects

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

Optimizations

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

2.5.2 NRMI in the J2EE Application Server Environment

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

automatic transformation of regular Java classes into EJBs, as, for example, in our GOTECH framework described in Chapter III.

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

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

parameters of a remote method are to be passed by copy-restore. The following code example shows how the programmer can use this new annotation.

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

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

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

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

  Object[] arguments = invocation.getArguments();

  //create linear map representations
  //for copy-restore arguments
  Object[][] linearRepresentations =
              NRMI.createLinearRepresentations(arguments);
  ...
```

```
//pass the invocation to the next interceptor in the chain
InvocationResponse response =
                            getNext().invoke(invocation);

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

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

return response;

}



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

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

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

response.addAttachment(LINEAR_MAP, linearReps);

return response;

}
```

2.5.3 Introducing NRMI through Bytecode Engineering

In some development environments, the programmer could find beneficial the ability to use the call-by-copy-restore semantics on top of a standard unmodified middleware implementation, supporting only the standard call-by-copy semantics. Furthermore, that environment might not provide any built-in facilities for flexible functionality enhancement such as interceptors. For example, the J-Orchestra automatic partitioning system, described in Chapter IV, has as one of its primary design objectives the ability to execute

partitioned programs using a standard RMI middleware implementation. By default, J-Orchestra uses the RMI call-by-reference semantics (remote reference) to emulate a shared address space for the partitioned programs. However, as Figure 2-3 shows, any access to a remote object through a remote reference incurs network overhead. Therefore, a program partitioned with J-Orchestra can derive substantial performance benefits by using the call-by-copy-restore semantics in some of its remote calls. It is exactly for these kind of scenarios that we developed our approach for introducing NRMI by retrofitting the bytecodes of application classes that use the standard RMI API.

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

2.5.3.1 User View: NRMIzer

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

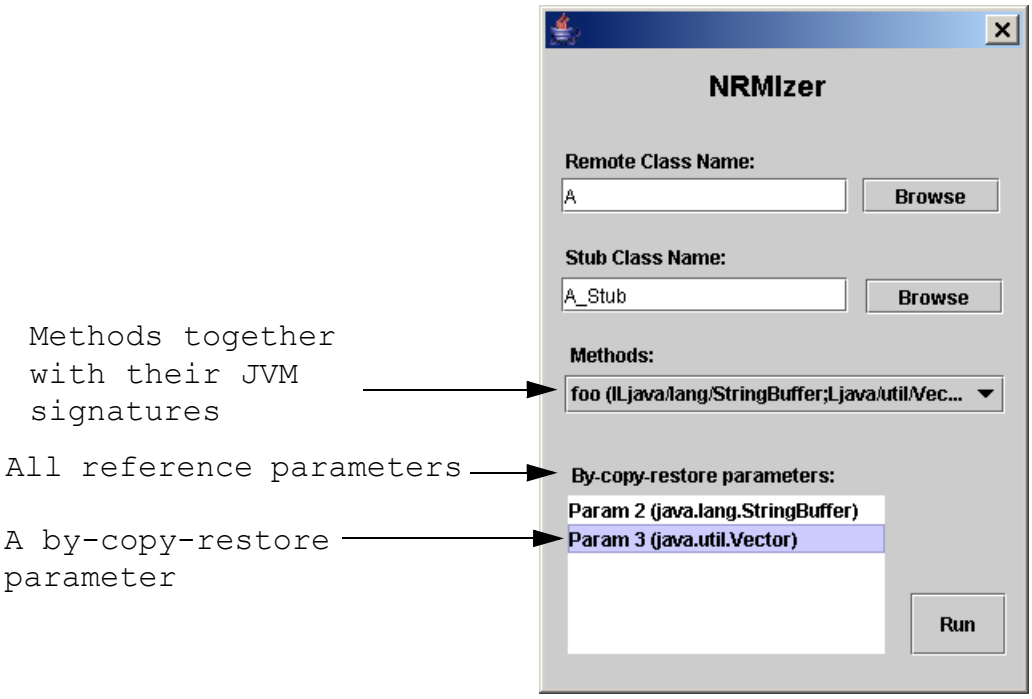


Figure 2-10: NRMIzer GUI.

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

2.5.3.2 Implementation Specifics: Backend Engine

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

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

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

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

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

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

2.6 Conclusion

In this chapter, we presented NRMI, a middleware mechanism that provides a fully-general implementation of call-by-copy-restore semantics for arbitrary linked data structures, used as parameters in remote procedure calls. We discussed the effects of calling

semantics for middleware, explained how our algorithm works, and described three different implementations of call-by-copy-restore middleware. In Chapter V we further discuss various applicability issues of NRMI, present several examples of Java programs in which NRMI is more convenient to use than regular Java RMI, and present detailed performance measurements of our drop-in RMI replacement implementation, proving that NRMI can be implemented efficiently enough for real world use.

CHAPTER III

GOTECH

This chapter describes GOTECH, a framework that can be used with a large class of unaware applications to turn their objects into distributed objects with minimal programming effort. GOTECH combines domain-specific and domain-independent tools to "aspectize" the distributed character of server-side applications to a much greater extent than with prior efforts. Specifically, the GOTECH framework has been developed on top of three main components: AspectJ (a high-level aspect language), XDoclet (a low-level aspect language), and NRMI (a middleware facility that makes remote calls behave more like local calls). We discuss why each of the three components offers unique advantages and is necessary for an elegant solution, why the GOTECH approach is general, and how it constitutes a significant improvement over past efforts to isolate distribution concerns.

3.1 Introduction

GOTECH (for "**G**eneral **O**bject **T**o **E**JB **C**onversion **H**elper") is a general framework for separating distribution concerns from application logic in enterprise Java applications via a mixture of aspect-oriented techniques and domain-specific tools. Following the objective of removing low-level technical barriers to the separation of distribution concerns, the framework operates under the assumption that the structure of the application is amenable to adding distribution. GOTECH targets the specific technical substrate of

server-side Java applications as captured by the J2EE specification [78]. This domain is technically challenging (due to complex conventions) and has been particularly important for applied software development in the last decade.

This work demonstrates how a combination of three tools can yield very powerful separation of distribution concerns in a server-side application. We call this separation "aspectization," following other aspect-oriented work. (We use the main aspect-oriented programming terms in this chapter, but do not embrace the full terminology. E.g. we avoid the AOP meaning of the term "component" as a complement of "aspect" [40].)

To classify the GOTECH approach, we can distinguish between three levels of aspectization of a certain concern or feature:

- ***Type 1: "out-of-sight"***. The application already exhibits the desired feature. The challenge of aspectization is to remove the relevant code and encapsulate it in a different entity (*aspect*) that is composable with the rest of the code at will. The approach is application-specific.
- ***Type 2: "enabling"***. The application does not exhibit the desired feature, but its structure is largely amenable to the addition of the feature. Code implementing the feature needs to be added in a separate aspect, but glue code may also need to be written to adapt the application logic and interfaces to the feature.
- ***Type 3: "reusable mechanism"***. Both the feature implementation and the glue code are packaged in a reusable entity that can be applied to multiple applications. Adapting an existing application to include the desired feature is trivial (e.g., a few annotations at the right places).

The GOTECH framework achieves Type 3 aspectization for a large class of server-side applications. In contrast, the closest prior work [72] attempts Type 1 aspectization and identifies several difficulties with the tools used: the need to write code to synchronize

views, the need to create application-specific interfaces for redirecting calls, and some others. GOTECH resolves these difficulties automatically. To achieve its goals, it uses three tools:

- ***NRMI*** [88]: a middleware mechanism described in detail in the previous chapter. NRMI is the key for going from a Type 1 aspectization to a Type 2. That is, it provides the mechanism for enabling an application that is written without distribution in mind to be distributed without significant changes to its logic. The NRMI semantics is indistinguishable from local execution for a large class of applications—e.g. all applications with single-threaded clients and stateless servers.
- ***AspectJ*** [41]: a high-level aspect language. It is used as a back-end, i.e. our framework generates AspectJ code. It eliminates a lot of the complexity of writing glue code to turn regular Java objects into Enterprise Java Beans (*EJB*s) [78].
- ***XDoclet*** [103]: a low-level aspect language. It is used primarily for generating the AspectJ glue code that adapts the application to the conventions of the distribution middleware. Like AspectJ, XDoclet is a widely available tool and our framework just provides XDoclet templates for our task. XDoclet is the key for going from a Type 2 aspectization to a Type 3. That is, it lets us capture the essence of the rewrite in a reusable template, applicable to multiple applications.

As an example (see Chapter V for a detailed description), we used GOTECH to turn an existing scientific application (a thermal plate simulator) into a distributed application. The application-specific code required for the distribution consists of only a few lines of annotations. The GOTECH framework provides the rest of the distribution-specific code.

3.2 The Elements of Our Approach

3.2.1 NRMI

The issue of reproducing the changes introduced by remote calls is important in aspectizing distribution. For instance, Soares et al. write in [72]:

> *When implementing the client-side aspect we had also to deal with the synchronization of object states. This was necessary because RMI supports only a copy parameter passing mechanism ...*

and

> *[Reproducing remote changes] requires some tedious code to be written ...*

Our NRMI middleware, described in detail in Chapter II, succeeds in making remote calls resemble local calls for many practical scenarios. For example, in the common case of a single-threaded client (multiple clients may exist but not as threads in the same process) and a stateless or memory-less server, NRMI calls are indistinguishable from local calls. With NRMI, the need for writing explicit code to reproduce remote changes is mostly eliminated. Thus, our approach can be more easily applied to unaware applications.

3.2.2 AspectJ

AspectJ [41] is a general purpose, high-level, aspect-oriented tool for Java. AspectJ allows the user to define aspects as code entities that can then be merged (*weaved*) with the rest of the application code. The power of AspectJ comes from the variety of changes it allows to existing Java code. With AspectJ, the user can add superclasses and interfaces to existing classes and can interpose arbitrary code to method executions, field references,

exception throwing, and more. Complex enabling predicates can be used to determine whether code should be interposed at a certain point. Such predicates can include, for instance, information on the identity of the caller and callee, whether a call to a method is made while a call to a certain different method is on the stack, and so forth.

For a simple example of the syntax of AspectJ, consider the code below:

```
aspect CaptureUpdateCallsToA {
  static int num_updates = 0;

  pointcut updates(A a): target(a) &&
                         call(public * update*(..));

  after(A a): updates(a) { // advice
    num_updates++; // update was just performed
  }
}
```

The above code defines an aspect that just counts the number of calls to methods whose name begins with "update" on objects of type `A`. The "pointcut" definition specifies where the aspect code will tie together with the main application code. The exact code ("advice") will execute after each call to an "update" method.

3.2.3 XDoclet

XDoclet is a widely used, open-source, extensible code generation engine [103]. XDoclet is often used to automatically generate wrapper code (especially EJB-related) given the source of a Java class. XDoclet works by parsing Java source files and meta-data (annotations inside Java comments) in the source code. Output is generated by using XDoclet template files that contain XML-style tags to access information from the source code. These tags effectively define a low-level aspect language. For instance, tags include `forAllClassesInPackage`, `forAllClassMethods`, `methodType`, and so forth.

XDoclet comes with a collection of predefined templates for common tasks (e.g., EJB code generation). Writing new templates allows arbitrary processing of a Java file at the syntax level. Creating new annotations effectively extends the Java syntax in a limited way.

3.3 The Framework

3.3.1 Overview

The GOTECH framework offers the programmer an annotation language[1] for describing which classes of the original application need to be converted into EJBs [78] and how (e.g., where on the network they need to be placed and what distribution semantics they support). The EJBs are then generated and deployed in an *application server*: a run-time system taking care of caching, distribution, persistence, and so forth of EJBs. The result is a server-side application following the J2EE specification [78]—the predominant server-side standard.

The importance of using EJBs as our distribution substrate is dual. First, it is the most mature technology for server-side development, and as such it has practical interest. Second, it has a higher technical complexity than middleware such as RMI. Thus, we show that our approach is powerful enough to handle near-arbitrary technical complications—our aspectization task is significantly more complex than that of [72] in terms of low-level interfacing.

Converting an existing Java class to conform to the EJB protocol requires several changes and extensions. An EJB consists of the following parts:

1. The annotations are introduced in Java source comments as “JavaDoc tags”. We use the term “annotation” instead of the term “tag” as much as possible to prevent confusion with the XDoclet “tags”, i.e. the XDoclet aspect-language keywords, like `forAllClassMethods`.

- the actual bean class implementing the functionality
- a home interface to access life cycle methods (creation, termination, state transitions, persistent storing, and so forth)
- a remote interface for the clients to access the bean
- a deployment descriptor (XML-formatted meta-data for application deployment).

In our approach this means deriving an EJB from the original class, generating the necessary interfaces and the deployment descriptor and finally redirecting all the calls to the original class from anywhere in the client to the newly created remote interface. The process of adding distribution consists of the following steps:

1. The programmer introduces annotations in the source
2. XDoclet processes the annotations and generates the aspect code for AspectJ
3. XDoclet generates the EJB
4. XDoclet generates the EJB interface and deployment descriptor
5. The AspectJ compiler compiles all generated code (including regular EJB code and AspectJ aspect code from step 1) to introduce distribution to the client by redirecting all client calls to the EJB instead of the original object.

(The XDoclet templates used in step 4 are among the pre-defined XDoclet templates and not part of the GOTECH framework.)

3.3.2 Framework Specifics

We discuss many of the technical specifics of GOTECH in this section. Further examples can be found in Chapter V, in which we present an example application.

3.3.2.1 Middleware

In our development we used the JBoss open-source application server. JBoss is one of the most widely used application servers with 2 million downloads in 2002. Although our approach would work with other application servers, they would need to somehow integrate NRMI. (An alternative discussed in Section 3.3.3 is to have XDoclet insert the right NRMI code in the application. This just changes the packaging of the code but not the need for NRMI, and it is technically much more convoluted.) Section 2.5.2 of this dissertation describes in detail the integration of NRMI in the JBoss code base as a middleware option. GOTECH uses NRMI just like any other client would.

3.3.2.2 GOTECH Annotations

In our approach, the programmer needs to provide annotations to guide the automated transformation process. Some of these annotations are EJB-specific (i.e. processed by existing XDoclet templates). Additionally, we added annotations for making remote calls use NRMI. Integrating copy-restore semantics required an extension of the JBoss-specific deployment descriptor. For instance, the following annotations will make a parameter passed using call-by-copy-restore. (This is a per-method annotation.)

```
/**
 * @ejb:interface-method view-type="remote"
 * @jboss:method-parameters copy-restore="true"
 */
```

Note that without invoking GOTECH the comments remain completely transparent to the original application.

3.3.2.3 GOTECH XDoclet Templates

After the programmer supplies all the necessary information, we can use XDoclet to generate files. The first task XDoclet is used for is creating the source code for the client aspect. The generated aspect's role is to redirect all method calls to the original objects to now be performed on the appropriate EJB. Additionally, the original object should only be referred to through an interface and its creation should be done by a distributed object factory instead of through the operator `new`. (We ignore direct field reference for now, but it could be handled similarly using AspectJ constructs.) A simplified (shorter XML tags, elided low-level details) fragment of our XDoclet template appears in Figure 3-1. The template file consists of plain text, in this case a basic AspectJ source file structure, and the XDoclet annotation parameters, whose value is determined by running XDoclet.

For ease of reference we have split the template in Figure 3-1 in three parts. Part I defines that the aspect is per-target, i.e. that a unique instance of the aspect will be created every time a target object (i.e. an instance of class `className`, which is derived from the `name` XDoclet parameter) is created. The other conditions in Part I determine that the interception of the construction of a target object should only occur if this takes place outside the control flow of the Aspect itself. Note that the template uses XDoclet's ability to access class information (`<className/>`) in addition to user-supplied annotations.

Part II of the template shows the code that will be executed for the creation of a new instance of the aspect. This is the code that takes care of the remote creation of the EJB using a remote object factory mechanism.

```
public aspect GOTECH_<className/>WrapperAspect
      pertarget(target(<className/>)
      && (!cflow(within(GOTECH_<className/>WrapperAspect)))) {

// Part I above: per-target aspect that captures object creation.

private
<classTagValue tagName="ejb:bean" paramName="interface-name"/> ep;

GOTECH_<className/>WrapperAspect() {
 try {
  <classTagValue tagName="ejb:bean" paramName="name"/>Home sh;
  javax.naming.InitialContext initContext =
                              new javax.naming.InitialContext();
  String JNDIName =
       "<classTagValue tagName="ejb:bean" paramName="jndi-name"/>";
  Object obj = initContext.lookup(JNDIName);
  sh = (<classTagValue tagName="ejb:bean" paramName="name"/>Home)
        javax.rmi.PortableRemoteObject.narrow( obj,
    <classTagValue tagName="ejb:bean" paramName="name">Home.class);
  ep = sh.create();
 } catch (Exception e) { ... }
}

// Part II above: Intercepting object creation.
// A remote object factory is called. All access is through an interface.

Object around() : target(<className/>)
              && call(* *(..))
              && (!cflow(within(GOTECH_<className/>WrapperAspect)))
{
 try {
  Method meth = ep.getClass().getMethod(
                thisJoinPoint.getSignature().getName(),
                ((org.aspectj.lang.reflect.MethodSignature)
                thisJoinPoint.getSignature()).getParameterTypes());
  Object result = meth.invoke(ep, thisJoinPoint.getArgs());
  return result;
 } catch (Exception e) { ... }
}

  // Part III above: Intercepting method calls.
}
```

Figure 3-1: Simplified fragment of XDoclet template to generate the aspect code. Template parameters are shown emphasized. Their value is set by XDoclet based on program text or on user annotations in the source file.

Finally, Part III makes the generated aspect code capture all method calls (`call(* *(..))`) to objects of class `className` unless the calls come from within the Aspect itself.

The next task for XDoclet is to transform the existing class into a class conforming to the EJB protocol. To do this, we need to make the class implement the `SessionBean` interface. Additionally, all parameters of methods of an EJB must implement interface `Serializable`: a Java marker interface used to designate that the parameter's state can be "pickled" and transported to a remote site. We do this by creating an aspect that when run through AspectJ will make the parameter types implement interface `Serializable`. The template file for this transformation is not shown, but the functionality is not too complex.

The last task in which we employ XDoclet is the generation of the home and remote interface as well as the deployment descriptors. XDoclet has predefined templates for this purpose. The only extension has to do with the copy-restore semantics and generating the right deployment descriptor to use NRMI. Note that this step needs to iterate over all methods of a class and replicate them in a generated interface, while adding a `throws RemoteException` clause to every method signature. This is a task that Soares et al. [72] had to perform manually in their effort to aspectize distribution with AspectJ. A simplified fragment of the XDoclet template for iterating over the methods appears below:

```
<forAllMethods>
 <ifIsInterfaceMethod interface="remote">
    public <methodType/> <methodName>
                          (<parameterList/> )
           <exceptionList append=
                   "java.rmi.RemoteException"/>;
 <ifIsInterfaceMethod>
</forAllMethods>
```

3.3.3 Discussion of Design

Our approach uses a combination of AspectJ, NRMI and XDoclet in order to add distribution to existing applications. Each tool has unique advantages and greatly simplifies our task. Of course, in terms of engineering choices, there are alternative approaches:

- instead of our three tools, we could have a single, special-purpose tool, like D [52], JavaParty [66] or AdJava [23] that will rewrite existing Java code and introduce new code and meta-data. (None of these tools deals with the EJB technology, but they are representatives of domain-specific tools for distribution.) We strongly prefer the GOTECH approach over such a "closed" software generator approach. The first reason is the use of widely available tools (AspectJ, XDoclet) that allow exposing the logic of the rewrite in terms of templates. Templates are significantly easier to understand and maintain than the source code of a compiler-level tool. The second advantage of our approach is the use of unobtrusive annotations inside Java source comments. That is, the annotations do not affect the source code of the original Java program, which can still be used in its centralized form.
- we could have XDoclet generate all the code, completely replacing both NRMI and AspectJ. In the case of NRMI, this would mean that XDoclet will act as an inliner/specializer: the NRMI logic would be added to the program code, perhaps specialized as appropriate for the specific remote call. Conceptually, this is not a different approach (the copy-restore semantics is preserved) but in engineering terms it would add a lot of complexity to XDoclet templates. Similarly, one can imagine replacing all uses of AspectJ with more complex XDoclet templates. Yet AspectJ allows manipulations taking Java semantics into account—e.g. the `cflow` construct mostly used for recogniz-

ing calls under the control flow of another call (i.e. while the latter is still on the execution stack). Although the emulation of this construct with a run-time flag is not too complex conceptually, it does require essentially replicating the functionality of AspectJ in a low-level, inconvenient, and hard-to-maintain way. XDoclet is not designed for such complex program manipulations.

- finally, one could ask whether a combination of AspectJ and NRMI without XDoclet would be sufficient. Unfortunately, this approach would suffer a more severe form of the drawbacks identified by Soares et al. [72]. These drawbacks include needing to write the remote interface code by hand, not being able to work without availability of source code, and so forth. The problem is exacerbated in our case because our target platform (EJBs) is more complex and because we are attempting complete automation. To automate the construction of EJBs, we need to generate the remote and home interfaces from the original class, as well as generate non-code artifacts (the deployment descriptor meta-data in XML form). None of these activities could be automatically handled by AspectJ. In general, low-level generation, like iterating over all methods and replicating them (with minor changes) in a new class or interface, is impossible with AspectJ. The same is true for "destructive" changes, like adding a `throws` clause to existing methods.

3.4 Advantages and Limitations

3.4.1 Advantages of our approach

Despite the simplicity of applying GOTECH, the resulting code is feature-by-feature analogous to that written manually by Soares et al. [72]. We discuss each element of the implementation and perform a comparison.

Making the object remote. With GOTECH, this step is quite simple. A new remote interface is created from the original class using XDoclet. Soares et al. identified several problems when trying to perform the same task with AspectJ, even though their original application already supported reference to the relevant objects through an interface. Specif-

ically, Soares et al. could not add a `RemoteException` declaration to the constructor of their "facade" class using AspectJ. In our approach, the original class does not need to be modified: a slightly altered copy forms the bean part of the EJB. It is easy to add exception declarations when the new class gets created (see the `exceptionList append` statement in Section 3.3.2).

Serializing types. Soares et al. needed to write by hand (listing all affected classes!) the aspect code that will make application classes extend the `java.io.Serializable` interface so they can be used as parameters of a remote method. In their paper, they acknowledge:

> *This might indeed be repetitive and tedious, suggesting that either AspectJ should have more powerful metaprogramming constructs or code analysis and generation tools would be helpful for better supporting this development step.*

Indeed, our approach fulfills this need. Using XDoclet, we create automatically the aspect code to make the parameter types implement `java.io.Serializable`.

Client call redirection. The code introduced by the generated aspect of Figure 3-1 (part III) does a similar redirection as with the technique of Soares et al. That is, it executes a call to the same method, with the same arguments, but with a different target (a remote interface instead of the original local reference). Nevertheless, in the Soares et al. technique this code had to be introduced manually for each individual method. These authors admit:

> *... [T]his solution works well but we lose generality and have to write much more tedious code. It is also not good with respect to software maintenance: for every new facade method, we should write an associated advice....*

We should note that it is not really XDoclet or NRMI that give us this advantage over the Soares et al. approach. Instead, our aspect code of Figure 3-1 (part III) uses Java reflection to overcome the type incompatibilities arising with a direct call. This technique is also applicable to the Soares et al. approach.

Updating Remotely Changed Data. NRMI offers a very general way to update local data after a remote method changes them. Our approach is not only more general than the one used by Soares et al. but also more efficient. Specifically, Soares et al. admit the need to "synchronize object states." They perform this task by trapping every call to an update method, storing the affected objects in a data structure, and eventually iterating over this data structure on the remote site and reproducing all the introduced changes. NRMI is a more general version of this technique, applicable to a large class of applications. The Health Watcher system of Soares et al. is one of them: the system is "non-concurrent" (as characterized by the authors) and the two sites do not need to always maintain consistent copies of data: it is enough to reproduce changes introduced by a remote call. Soares et al. acknowledge both the need for automation and the fact that the structure of state synchronization in Health Watcher is general:

> *... it would be helpful to have a code analysis and generation tool that would help the programmer in implementing this aspect for different systems complying to the same architecture of the Health Watcher system.*

Additionally, NRMI is more efficient than capturing all calls to update methods. Instead of intercepting every update call, NRMI allows the remote call to proceed at full speed and only after the end of its execution it collects the changed data. (To do this, before execution of the remote call, NRMI needs to store pointers to all data reachable by param-

eters. This is not costly, since these data are transferred over the network anyway.) Soares et al. admit the inefficiency of their approach, although they argue it does not matter for the case of Health Watcher.

3.4.2 Limitations

Currently the GOTECH framework suffers from some engineering limitations. We outline them below. Some of these limitations are shared by the approach of Soares et al.—assuming that this approach is applied to multiple applications. Recall, however, that our templates only automate some tedious tasks. Although these templates are not application-specific, they also do not attempt complete coverage for all Java language features. In general, it is up to the programmer to ensure that the GOTECH process is applicable to the application.

3.4.2.1 Entity Bean support

So far we have only concentrated on distributing the computation of an application. Thus, we only have templates for generating Session Beans and not Entity Beans. Entity Beans are commonly used for representing database data through an object view. There is no further technical difficulty in producing templates for Entity Beans, but their value is questionable in our case. First, we are not aware of an example where adding distribution to an existing application requires creating any Entity Beans. Second, the Entity Bean generation will have more constraints than Session Beans—for instance, Entity Beans should support identity operations (retrieval by primary key) since they are meant for use with databases. These operations usually cannot be supplied automatically—the original class

will have to support such operations, or a fairly complex XDoclet annotation could supply the needed information.

3.4.2.2 Conditions for applying rewrite

Our aspect code controlling where we apply indirection in the original code is currently coarse grained. Consider again Part I of Figure 3-1 The generated aspect code is applied everywhere except in points in the execution under the control flow of the EJB. This roughly means that our approach assumes that the desired distributed application is split into a client site and a server site, and the server site never calls back to the client. On the server site, the calls to the existing class are not redirected. The positive side-effect of this rule is that server-side objects communicate with each other directly, thus suffering no overhead. Future versions could have a finer grained control over when the indirection should be applicable.

3.4.2.3 Making types serializable

Our current approach of making classes implement `java.io.Serializable` so that they could be passed as parameters to remote method calls works only for some application classes. Indeed, our current XDoclet template for generating aspects that adds `java.io.Serializable` to all non-serializable parameter types makes several assumptions.

One assumption is that having a type implement this marker interface is sufficient for making it serializable by Java Serialization [79]. However, this is not always the case: a type is serializable only if all the types reachable transitively from it are also serializable. Our current implementation performs no such check. Nevertheless, this is a reasonable

assumption for a framework that assumes that the original centralized application is amenable for distribution in the first place. Making a type adhere to the serializability requirements could be non-trivial, requiring significant changes to its implementation. In that case, careful manual code restructuring often is the only feasible option for performing these changes.

Another assumption is that all non-serializable parameters are application classes. In other words, all system JDK classes, passed as parameters to a remote method, must be serializable, for applying aspects to system classes has not been standardized. Even if changing the implementation of a system class (i.e., having it implement an additional interface) were straightforward, that would violate the design principles of our framework, creating the need for a custom runtime environment. However, in practice it never makes sense to modify the serializability properties of a system class. Because significant effort has gone into designing system classes that are part of the standard JDK, the ones that are serializable are always marked as such (i.e., implementing `java.io.Serializable`).[2] The system classes that are not serializable are usually the ones that control some local system resources such as threads, sound, etc. It would be meaningless to send instances of such classes over the network anyway. Thus, making a centralized program amenable to our approach requires restructuring it in such a way that no non-serializable system classes are used as parameters in remote methods.

2. In addition, instances of some serializable system classes could become invalid if serialized and transferred to a machine on a different network node. E.g., `java.io.File`.

3.4.2.4 Exceptions, construction, field access

The current state of our templates leaves some more minor engineering issues unresolved. For instance, the handling of remote method exceptions is generic and cannot be influenced by the programmer at this stage. This is just a matter of regular Java programming: we need to let user code register exception handlers that will get called from the `catch` clauses of our generated code. Another shortcoming of our template of Figure 3-1 is that it only supports zero-argument constructors. (This is fine for stateless Session Beans, which by convention have no-argument constructors.) However, it would be only a matter of engineering to implement an additional rewrite to address this problem. We also currently have no support for adding indirection to direct field access from the client object to the remote object, which should be quite feasible with AspectJ. Nevertheless, direct access to fields of another object may mean that the two objects are tightly coupled, suggesting that perhaps they should not be split in the distributed version.

The subset of implemented functionality in the current version of GOTECH is sufficient to illustrate our approach. At the same time, all of the remaining issues would be relatively easy to address in a production system—GOTECH templates are highly amenable to inspection and modification. In fact, it is quite feasible that application programmers would incorporate additional functionality to GOTECH on a per-application basis.

Finally, since performance is an important concern, we should emphasize that it is not an issue for the GOTECH framework. For the most part, GOTECH just generates the code that a programmer would otherwise add by hand. Additionally, in the only case in which something is done automatically (when using NRMI) the mechanism is quite optimized [88]. In general, however, for a given set of distribution and caching decisions, the

constant computational overheads of a distribution mechanism like ours are relatively unimportant. These overheads are small relative to the inherent cost of communication (including network time and middleware, e.g., EJB, overheads). These costs are not important if only few objects are accessed remotely. On the other hand, if many objects are accessed remotely, any distribution mechanism will suffer.

3.5 Conclusions

We presented the GOTECH framework: an approach to aspectizing distribution concerns. GOTECH relieves the programmer from performing many of the tedious tasks associated with distribution. GOTECH relies on NRMI: a middleware implementation that makes remote calls behave much like local calls for a large class of uses (e.g. single-threaded access to client data and no memory of past call arguments on the server). Additionally, GOTECH only depends on general-purpose tools and offers an easy to evolve implementation, easily amenable to inspection and change. Compared with the closest past approaches, GOTECH is significantly more convenient and general.

In high-level terms, GOTECH is also interesting as an instance of a collaboration of generative and aspect-oriented techniques. The generative elements of GOTECH are very simple exactly because AspectJ handles much of the complexity of where to apply transformations and how. On the other hand, AspectJ alone would not suffice to implement GOTECH.

Section 5.6 presents an example of applying the GOTECH framework to convert a centralized scientific application into a distributed application interacting with an application server.

CHAPTER IV

J-ORCHESTRA

This chapter presents J-Orchestra, an automatic partitioning system for Java programs. J-Orchestra takes as input a Java program in bytecode format and transforms it into a distributed application, running across multiple Java Virtual Machines (JVMs). To accomplish such automatic partitioning, J-Orchestra substitutes method calls with remote method calls, direct object references with proxy references, and so forth, by means of bytecode rewriting and code generation. The partitioning does not involve any explicit programming or modifications to the JVM or its standard runtime classes. The main novelty and source of scalability of J-Orchestra is its approach to dealing with unmodifiable code (e.g., Java system classes). The approach consists of an analysis heuristic that determines which application objects get passed to which parts of native (i.e., platform-specific) code and a technique for injecting code that will convert objects to the right representation so that they can be accessed correctly inside both application and native code. Validating the type information accuracy and testing the correctness of the analysis heuristic have demonstrated its viability in the J-Orchestra context. To be able to run partitioned programs over a standard remote procedure call middleware such as RMI, J-Orchestra introduces a new approach to maintaining the Java centralized concurrency and synchronization semantics over RMI efficiently. Finally, specialized domains present opportunities for making J-Orchestra partitioning more automatic, which is the case for appletizing—a semi-automatic

approach to transforming a Java GUI application into a client-server application, in which the client runs as a Java applet that communicates with the server through RMI.

4.1 Introduction

Adding distributed capabilities to existing programs has come to the forefront of software evolution [44] and is commonly accomplished through application partitioning—the task of splitting up the functionality of a centralized monolithic application into distinct entities running across different network sites. As a programming activity, application partitioning entails re-coding parts of the original application so that they could interact with a distributed middleware mechanism such as Remote Procedure Call (RPC) [10] or Common Object Request Broker Architecture (CORBA) [61]. In general, this manual process is costly, tedious, error prone, and sometimes infeasible due to the unavailability of source code, as in the case of many commercial applications.

Automating, even partially, a tedious and error-prone software development task is always a desirable goal. Thus, automating application partitioning would not only save programming time but would also result in an effective approach to separating distribution concerns. Having a tool that under human guidance handles all the tedious details of distribution could relieve the programmer of the necessity to deal with middleware directly and to understand all the potentially complex data sharing through pointers.

Automating any programming task presents an inherent dichotomy between power and automation: any automation effort hinders complete control for users with advanced requirements. Indeed, transforming a centralized application for distributed execution often requires changes in the logic and structure of the application to satisfy such requirements

as fault tolerance, load balancing, and caching. In view of this dichotomy, one important question is what kind of common architectural characteristics make applications amenable to automatic partitioning, and when meaningful partitioning is impossible without manually changing the structure and logic of the application first.

J-Orchestra operates on binary (Java bytecode) applications and enables the user to determine object placement and mobility to obtain a meaningful partitioning. The application is then re-written to be partitioned automatically and different parts can run on different machines, on unmodified versions of the Java VM. For a large subset of Java, the resulting partitioned application's execution semantics is identical to the one of its original, centralized version. The requirement that the VM not be modified is important. Specifically, changing the runtime is undesirable both because of deployment reasons (it is easy to run a partitioned application on a standard VM) and because of complexity reasons (Java code is platform-independent, but the runtime system has a platform-specific, native-code implementation).

The conceptual difficulty of performing application partitioning in general-purpose languages (such as Java, C#, but also C, C++, etc.) is that programs are written to assume a shared memory: an operation may change data and expect the change to be visible through all other pointers (*aliases*) to the same data. The conceptual novelty of J-Orchestra (compared to past partitioning systems [33][75][84] and distributed shared memory systems [2][3][5][14][102]) consists of addressing the problems resulting from inability to analyze and modify all the code under the control flow of the application. Such unmodifiable code is usually part of the runtime system on which the application is running. In the case of Java, this runtime is the Java VM. In the case of free-standing applications, the runt-

ime is the OS. Without complete control of the code, execution is in danger of letting a reference to a remote object get to code that is unaware of remoteness. Prior partitioning systems have ignored the issues arising from unmodifiable code and have had limited scalability, as a result. J-Orchestra features a novel rewrite mechanism that ensures that, at run-time, references are always in the expected form ("direct" = local or "indirect" = possibly remote) for the code that handles them. The result is that J-Orchestra can split code that deals with system resources, safely running, e.g., all sound synthesis code on one machine, while leaving all unrelated graphics code on another.

This chapter starts by describing the general partitioning approach of J-Orchestra and its analysis algorithm and rewriting engine. Then it covers how J-Orchestra maintains the Java centralized concurrency and synchronization semantics over RMI efficiently. Finally, it demonstrates how specialized domains present opportunities to make J-Orchestra partitioning more automatic through the case of appletizing.

Chapter V of this dissertation identifies the environment features that make J-Orchestra possible and argues that partitioning systems following the principles laid out by J-Orchestra are valuable in modern high-level run-time systems such as the Java VM or Microsoft's CLR. Chapter V also presents several case-studies that demonstrate J-Orchestra handling arbitrary partitioning of realistic applications without requiring an understanding of their internals.

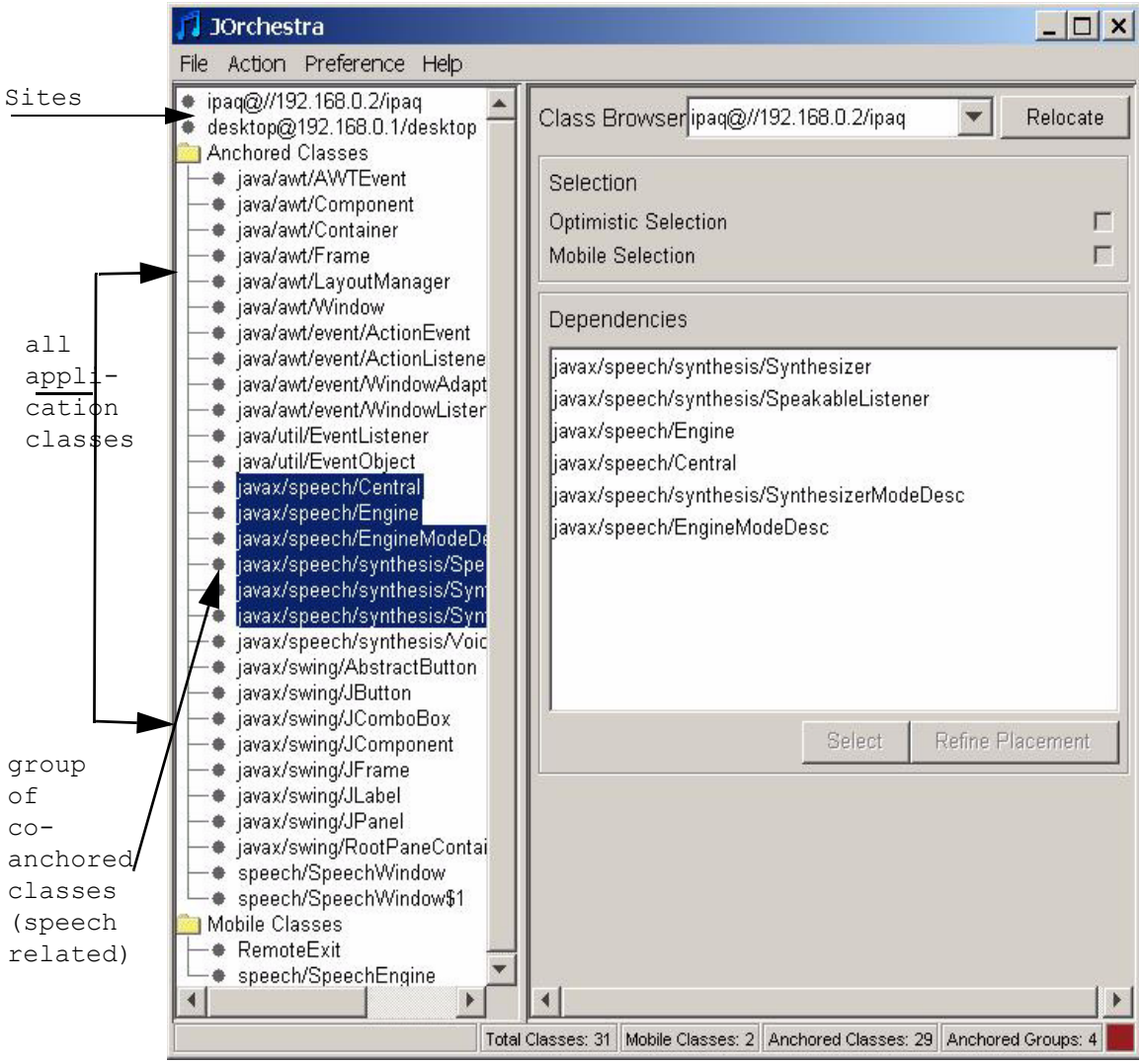


Figure 4-1: Example user interaction with J-Orchestra. An application controlling speech output is partitioned so that the machine doing the speech synthesis is different from the machine controlling the application through a GUI.

4.2 User View of J-Orchestra

Figure 4-1 shows a screenshot of J-Orchestra in the process of partitioning a small but realistic example application. The original example Swing application showcases the Java Speech API and works as follows: the user chooses predefined phrases from a drop-down box and the speech synthesizer pronounces them. As a motivation for partitioning,

imagine a scenario in which this application needs to be run on a small device such as a PDA that either has no speakers (hardware resource) or does not have the Speech API installed (software resource). The idea is to partition the original application in a client-server mode so that the graphical partition (i.e., the GUI), running on a PDA, would control the speech partition, running on a desktop machine. We chose this particular example because it fits well into the realm of applications amenable for automatic application partitioning. The locality patterns here are very clear and defined by the specific hardware resources (graphical screen and speech synthesizer) and their corresponding classes (Swing and Speech API).

Figure 4-1 shows J-Orchestra at a point when it has finished importing all the referenced classes of the original application and has run its classification algorithm (Section 4.4) effectively dividing them into two major groups represented by tree folders *anchored* and *mobile*.

- Anchored classes control specific hardware resources and make sense within the context of a single JVM. Their instances must run on the JVM that is installed on the machine that has the physical resources controlled by the classes. J-Orchestra clusters anchored classes into groups for safety; intuitively, classes within the same anchored group reference each other directly and as such must be co-located during the execution of the partitioned application. If classes from the same group are placed on the same machine, the partitioned application will never try to access a remote object as if it were local, which would cause a fatal run-time error. J-Orchestra classification algorithm (Section 4.4) has created four anchored groups for this example. One group contains all the referenced speech API classes. The remaining groups specify various Swing classes. While classes within the same anchored group cannot be separated, anchored groups can be placed on different network sites. In our example, all the Swing classes anchored groups should be

placed on the site that will handle the GUI of the partitioned application to obtain meaningful partitioning.

- Mobile classes do not reference system resources directly and as such can be created on any JVM. Mobile classes do not get clustered into groups, except as an optimization suggestion. Instances of mobile classes can move to different JVMs independently during the execution to exploit locality. Supporting mobility requires adding some extra code to mobile classes at translation time to enable them to interact with the runtime system. Mobility support mechanisms create overhead that can be detrimental for performance if no mobility scenarios are meaningful for a given application. To eliminate this mobility overhead, a mobile class can be *anchored by choice*. We discuss anchoring by choice and its implications on the rewriting algorithm in Section 4.5.2.

The J-Orchestra GUI represents each network node in the distributed application by a dedicated tree folder. The user then drag-and-drops classes from the anchored and mobile folders to their destination network site folder. Putting an anchored class in a particular network folder assigns its final location. For a mobile class, it merely assigns its initial creation location. Later, an instance of a mobile object can move as described by a given mobility policy. When all classes are assigned to destination folders, the J-Orchestra rewriting tool transforms the original centralized application into a distributed application. At the end, J-Orchestra puts all the modified classes, generated supporting classes, and J-Orchestra runtime configuration files into `jar` files, one per destination network site.

At run-time, J-Orchestra employs its runtime service to handle such tasks as remote object creation, object mobility, and various bookkeeping tasks.

4.3 The General Problem and Approach

In abstract terms, the problem that J-Orchestra solves is *emulating a shared memory abstraction for unaware applications without changing the runtime system*. The following two observations distinguish this problem from that of related research work. First, the requirement of not changing the run-time system while supporting unaware applications sets J-Orchestra apart from traditional Distributed Shared Memory (DSM) systems. (The related work chapter (Chapter VII) offers a more complete comparison.) Second, the implicit assumption is that of a pointer-based language. It is conceptually trivial to support a shared memory abstraction in a language environment in which no sharing of data through pointers (aliases) is possible. Although it may seem obvious that realistic systems will be based on data sharing through pointers,[1] the lack of data sharing has been a fundamental assumption for some past work in partitioning systems—e.g., the Coign approach [33].

It is worth asking why mature partitioning systems have not been implemented in the past. For example, why no existing technology allows the user to partition a platform-specific binary (e.g., an x86 executable) so that different parts of the code can run on different machines? We argue that the problem can be addressed much better in the context of a high-level, object-oriented runtime system, like the JVM or the CLR, than in the case of a platform-specific binary and runtime. The following three concrete problems need to be overcome before partitioning is possible:

1. The pointers may be hidden from the end user (e.g., data sharing may only take place inside a Haskell monad). The problems identified and addressed by J-Orchestra remain the same regardless of whether the end programmer is aware of the data sharing or not.

1. The granularity of partitioning has to be coarse enough: the user needs to have a good vocabulary for specifying different partitions. High-level, object-oriented runtime systems, like the Java VM, help in this respect because they allow the user to specify the partitioning at the level of objects or classes, as opposed to memory words.

2. It is necessary to establish a mechanism that adds an indirection to every pointer access. This involves some engineering complexity, especially under the requirement that the runtime system remain unmodified.

3. The indirection has to be maintained even in the presence of unmodifiable code. Unmodifiable code is usually code in the application's runtime system. For example, in the case of a stand-alone executable running on an unmodified operating system, the program may create entities of type "file" and pass them to the operating system. If these files are remote, a runtime error will occur when they are passed to the unsuspecting OS. Addressing the problem of adding indirection in the presence of unmodifiable code is the main novelty of J-Orchestra. This problem, in different forms, has plagued not just past partitioning systems but also traditional Distributed Shared Memory systems. Even page-based DSMs often see their execution fail because protected pages get passed to code (e.g., an OS system call expecting a buffer) that is unaware of the mechanism used to hide remoteness.

We now look at the problem in more detail, in order to see the complications of adding indirection to all pointer references. The standard approach to such indirection is to convert all direct references to indirect references by adding proxies. This creates an abstraction of shared memory in which proxies hide the actual location of objects—the actual object may be on a different network site than the proxy used to access it. This abstraction is necessary for correct execution of the program across different machines because of *aliasing*: the same data may be accessible through different names (e.g., two different pointers) on different network sites. Changes introduced through one name/pointer

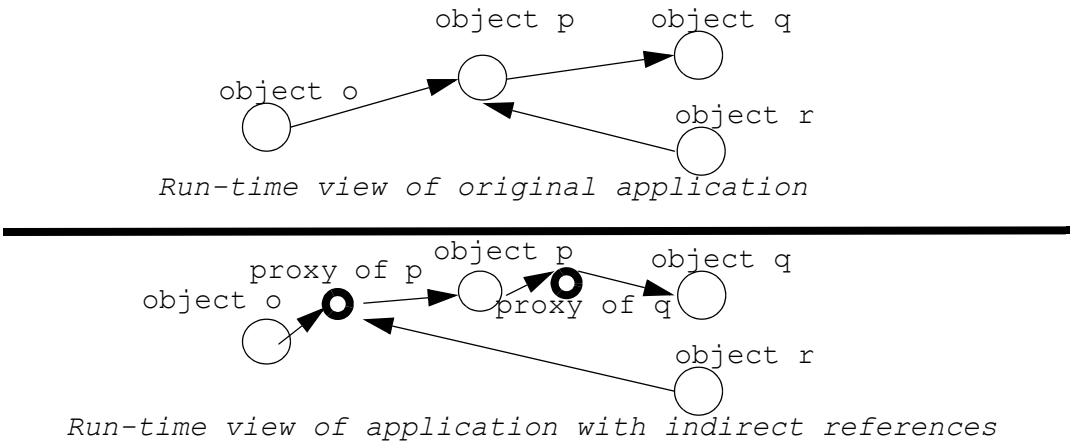


Figure 4-2: Results of the indirect reference approach schematically. Proxy objects could point to their targets either locally or over the network.

should be visible to the other, as if on a single machine. Figure 4-2 shows schematically the effects of the indirect referencing approach. This indirect referencing approach has been used in several prior systems [66][74][84].

Since one of our requirements is to leave the runtime system unchanged, we cannot change the JVM's pointer/reference abstraction. Instead, J-Orchestra rewrites the entire partitioned application to introduce proxies for every reference in the application. Thus, when the original application would create a new object, the partitioned application will also create a proxy and return it; whenever an object in the original application would access another object's fields, the corresponding object in the partitioned application would have to call a method in the proxy to get/set the field data; whenever a method would be called on an object, the same method now needs to be called on the object's proxy; etc.

The difficulty of this rewrite approach is that it needs to be applied to *all code that might hold references to remote objects*. In other words, this includes not just the code of the original application but also the code inside the runtime system. In the case of the Java

VM, such code is encapsulated by system classes that control various system resources through native code. Java VM code can, for instance, have a reference to a thread, window, file, etc., object created by the application. However, not being able to modify the runtime system code, one can not make it aware of the indirection. For instance, one cannot change the code that performs a file operation to make it access the file object correctly for both local and remote files: the file operation code is part of the Java VM (i.e., in machine-specific binary code) and partly implemented in the operating system. If a proxy is passed instead of the expected object to runtime system code that is unaware of the distribution, a run-time error will occur. Without changing the platform-specific runtime (JVM+OS) of the application, one cannot enable remoteness for all of the code.[2] (For simplicity, the implicit assumption is that the application itself does not contain native code—i.e., it is a "pure Java" application.)

J-Orchestra effectively solves many of the problems of dealing with unmodifiable code by partitioning *around* unmodifiable code. This approach consists of the following two parts. The first is the classification algorithm: a static analysis that determines which classes should be co-located. The second is the rewrite algorithm, which inserts the right code in the partitioned application so that, at run-time, indirect references are converted to direct and vice versa when they pass from mobile to anchored code. In order to perform classification, even though one cannot analyze the platform-specific binary code for every platform, J-Orchestra employs a heuristic that relies on the type information of the inter-

2. It is interesting to compare the requirements of adding indirection to those of a garbage collector. A garbage collector needs to be aware of references to objects, even if these references are manipulated entirely through native code in a runtime system. Additionally, in the case of a copying collector, the GC needs to be able to change references handled by native code. Nonetheless, being aware of references and being able to change them is not sufficient in our case: we need full control of all code that manipulates references, since the references may be to objects on a different machine and no direct access may be possible.

faces to the Java runtime (i.e., the type signatures of Java system classes). This is another way in which high-level, object-oriented runtime systems make application partitioning possible.

The end result is that, unlike past systems [66][74][84], J-Orchestra ensures safety while imposing a minimum amount of restrictions on the placement of objects in a pure Java application. Under the assumption that the classification heuristic is correct, which it typically is, the programmer does not need to worry about whether remote objects can ever get passed to unmodifiable code. Additionally, objects can refer to system objects through an indirection from everywhere on the network. If they need to ever pass such references to code that expects direct access, a direct reference will be produced at run-time.

Next we describe the three major technical components of J-Orchestra—the classification heuristic, the translation engine, and the handling of concurrency and synchronization.

4.4 Classification Heuristic

The J-Orchestra classification algorithm [87] classifies each class as anchored or mobile and determines anchored class groups. Classes in an anchored group must be placed on the same network site since they access each other directly.

The purpose of the classification algorithm is to determine the rewriting strategy that J-Orchestra must follow to enable the indirect referencing approach for each class in the partitioned application. In other words, classification informs the rewriter about generating and injecting code, as opposed to having the user specify this information manually. We have already described the first criterion of classification: each class can be either anchored

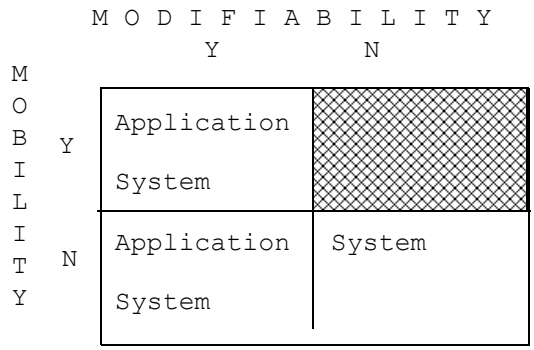


Figure 4-3: J-Orchestra classification criteria. For simplicity, we assume a "pure Java" application: no unmodifiable application classes exist.

or mobile. The second criterion deals with modifiability properties of a class: each class is either modifiable or not. A class is unmodifiable if its instances are manipulated by native code (e.g., if it has native methods or if its instances may be passed to native methods of other objects). Such dependencies inhibit the spectrum of changes one can make to the class's bytecode (sometimes none) without rendering it invalid. Figure 4-3 presents a diagram that shows all possible combinations of the classification criteria. As the diagram depicts, J-Orchestra distinguishes between three categories of classes: *mobile*, *anchored modifiable*, and *anchored unmodifiable*.

By examining the J-Orchestra classification criteria in Figure 4-3, one can draw several observations about the relationship between mobility and modifiability. One is that only a modifiable class can be rewritten so that its instances could participate in object mobility scenarios. I.e., unmodifiable mobile quadrant does not have any entries. Another observation is that only systems classes can be unmodifiable (in a "pure Java" application), and all unmodifiable systems classes are anchored. Finally, both application and systems classes can be mobile and modifiable.

Before presenting the rules that J-Orchestra follows to classify a class as unmodifiable, we demonstrate the idea informally through examples. Consider class `java.awt.Component`. This class is anchored unmodifiable because it has a native method `initIDs`. It is anchored because it must remain on the site of native platform specific runtime libraries on which it depends. It is unmodifiable because modifying its bytecode could render it invalid. As an example of a destructive modification, consider changing the class's name. Because the class's name is part of a key that matches native method calls in the bytecode to their actual native binary implementations, the class would no longer be able to call its native methods. A more general reason for not modifying the bytecode of an unmodifiable class is that because native code may be accessing directly the object layout (e.g., reading object fields). Having native methods, however, is not the only condition that could make it possible for instances of a class to be passed to native code. Consider class `java.awt.Point`, which does not have any native dependencies. However, `java.awt.Component` has a method `contains` that takes a parameter of type `java.awt.Point`. Because `java.awt.Component` is unmodifiable, its `contains` method can take only an instance of the original class `java.awt.Point` rather than its proxy—method `contains` could be accessing the fields of its `java.awt.Point` parameter directly. Therefore, if `java.awt.Point` is used in the same program (along with `java.awt.Component`), its classification category would be anchored as well. Furthermore, in the J-Orchestra methodology, we refer to such classes as *co-anchored*, meaning that because of the possibility of accessing each other directly, these classes must be kept together on the same site throughout the execution.

Conceptually, the classification heuristic has a simple task. It computes for each class *A* and *B* an answer to the question: *can references to objects of class A leak to unmodifiable (native) code of class B*? If the answer is affirmative, *A* cannot be remote to *B*: otherwise the unmodifiable code will try to access objects of class *A* directly (e.g., to read their fields), without being aware that it accesses an indirection (i.e., a proxy) resulting in a run-time error. This criterion determines whether *A* and *B* both belong to the same anchored group. If no constraint of this kind makes class *A* be part of an anchored group, and class *A* itself does not have native code, then it can be mobile. Next we present a heuristic, consisting of four basic rules, through which J-Orchestra co-anchors classes to anchored groups. Each co-anchored group must stay on the same site throughout the distributed execution. These rules essentially express a transitive closure, and the J-Orchestra classification iterates them until it reaches a fixed point.

1. Anchor a system class with native methods.

2. Co-anchor an anchored class with system classes used as parameters or return types of its methods or static methods.

3. Co-anchor an anchored class with the system class types of all its fields or static fields.

4. Co-anchor a system class, other than `java.lang.Object`, with its subclasses and superclasses.

The following few points are worth emphasizing about our classification heuristic:

- The above rules represent the essence of the analysis rather than its exhaustive description. The abbreviated form of the rules improves readability, especially since the analysis is based on heuristic assumptions, and therefore we do not make an argument of strict correctness.

- Specifically, the rules do not mention arrays or exceptions—these are handled similarly to regular classes holding references to the array element type and method return types, respectively. In addition, an array type is considered together with all its constituent types (e.g., an array type `T[][]` has constituent types `T[]` and `T`).
- Not all access to application objects inside native code/anchored classes is prohibited—only access that would break if a proxy were passed instead of the real object. Notably, the rules ignore Java interfaces: interface access from unmodifiable code is safe and imposes no restriction. Indeed, anchored unmodifiable code can even refer to mobile objects and to anchored objects in different groups through interfaces. The reason is that an interface does not allow direct access to an object: it does not create a name dependency to a specific class, and it cannot be used to access object fields. Because a proxy can serve just as well as the original object for access through an interface, distribution remains transparent for interface accesses.
- The rules codify a simple type-based heuristic. It computes all types that get passed to anchored code, based on information in the type signatures of methods and the calling information in the methods (in either application or system classes) that consist of regular Java bytecode. This is a conservative approach, as it only provides analysis on a per-type granularity and always assumes the worst: if an instance of class `A` can be passed to native code, all instances of any subtype of `A` are considered anchored (we make an explicit exception for `java.lang.Object`, or no partitioning would be possible).
- Despite the conservatism, however, the algorithm is not safe! The unsafety is inherent in the domain: no analysis algorithm, however conservative or sophisticated, can be safe if the unmodifiable code itself cannot be analyzed. The real data flow question we would like to ask is "what objects get accessed directly (i.e., in a way that would break if a proxy were used) by unmodifiable code?" The fully conservative answer is "all objects" since unmodifiable code can perform arbitrary side-effects and is not realistically analyzable, because it is only available in platform-specific binary form. Thus, unmodifiable code in the Java VM could (in theory) be accessing directly *any* object created by the application. For example, when an application creates a `java.awt.Component`, it is possible that some other, seemingly unrelated native system code, will maintain a ref-

erence to this object and later access its fields directly, preventing that code from running on a remote machine.

- In the face of inherent unsafety, our classification is an engineering approximation. We rely on the rich type interfaces and on the informal conventions used to code the Java system services. Specifically, we make the following three engineering assumptions about native code behavior. First, we assume that classes without native methods do not have specialized semantics (i.e., no object is accessed by unmodifiable code unless it is passed at some point in the program explicitly to such code through a native method call). This assumption also implies that all system objects are created under direct application control rather than spontaneously in the native code. Second, we assume that system classes's type information is strong, and that the system services do not discover type information not present in the type signatures (i.e., native code does not make assumptions about an `Object` reference passed to it by dynamically discovering the real type of the object and accessing its fields directly). Finally, we assume that native code does not share state between different pieces of functionality such as I/O, graphics, and sound (i.e., native code controlling different system resources are autonomous entities that can be safely separated to run on different JVMs).
- Although the assumptions of our classification heuristic are arbitrary, it is important to emphasize again that any different assumptions would be just as arbitrary: safety is impossible to ensure unless either partitioning is disallowed (i.e., a single partition is produced) or platform-specific native code can be analyzed. Since the classification analysis will be heuristic anyway, its success or failure is determined purely by its scalability in practice. We present empirical evidence on the accuracy of the first two assumptions in Chapter VI, and our experience of partitioning multiple applications that use different sets of native resources has confirmed the last assumption to be well-founded as well. As we discuss in Section 4.9, because our assumptions do not hold in certain cases (e.g., for `Thread` objects, or implicit objects like `System.in`, `System.out`), we provide specialized treatment for such objects.
- A more exact (less conservative) classification algorithm would be possible. For example, we could perform a data-flow analysis to determine which objects can leak to

unmodifiable code on a per-instance basis. The current classification heuristic, however, fits well the J-Orchestra model of type-based partitioning (recall that the system is semi-automatic and does not assume source code access: the user influences the partitioning by choosing anchorings on a per-class basis). Choosing a more sophisticated algorithm is orthogonal to other aspects of J-Orchestra. In particular, the J-Orchestra rewriting engine (Section 4.5) will remain valid regardless of the analysis used. In practice, J-Orchestra allows its user to override the classification results and explicitly steer the rewrite algorithm.

Our discussion so far covered modifiable and anchored unmodifiable classes, but left out *anchored modifiable classes*. The vast majority of these classes are not put in this category by the classification algorithm. Instead, these classes could be mobile, but are anchored by choice by the user of J-Orchestra. As briefly mentioned earlier, anchoring by choice is useful because it lets the class's code access all co-anchored objects without suffering any indirection penalty. Some of the anchored modifiable classes, however, are automatically classified as such by the classification heuristic. These classes are direct subclasses of anchored unmodifiable classes with which they are co-anchored. An application class `MyComponent` that extends `java.awt.Component` would be an example of such a class. This class does not have any native dependencies of its own, but it inherits those dependencies from its super class. As a result, both classes have to be co-anchored on the same site. Since `MyComponent` is an application class, it can support some limited bytecode manipulations. For example, it is possible to change bytecodes of individual methods or add new methods without invalidating the class. At the same time, changing `MyComponent`'s superclass would violate its intended original semantics. That is why J-Orchestra must follow a different approach to enable remote access to anchored modifiable classes.

4.5 Rewriting Engine

Having introduced and evaluated the J-Orchestra classification heuristic, we can now describe how the classification information gets used. The J-Orchestra rewriting engine is parameterized with the classification information. The classification category of a class determines the set of transformations it goes through during rewriting. The term "rewriting engine" is a slight misnomer due to the fact that applying binary changes to existing classes is not the only piece of functionality required to enable indirect referencing. In addition to bytecode manipulation,[3] the rewriting engine generates several supporting classes and interfaces in source code form. Subsequently, all the generated classes get compiled into bytecode using a regular Java compiler. We next describe the main ideas of the rewriting approach.

4.5.1 General Approach

The J-Orchestra rewrite first makes sure that all data exchange among potentially remote objects is done through method calls. That is, every time an object reference is used to access fields of a different object and that object is either mobile or in a different anchored group, the corresponding instructions are replaced with a method invocation that will get/set the required data.

For each mobile class, J-Orchestra generates a proxy that assumes the original name of the class. A proxy class has the same method interface as the original class and dynamically delegates to an implementation class. Implementation classes, which get generated by binary-modifying the original class, come in two varieties: *remote* and *local-only*. The

3. We use the BCEL library [18] for bytecode engineering.

difference between the two is that the remote version extends `UnicastRemoteObject` while the local-only does not. Subclasses of `UnicastRemoteObject` can be registered as RMI remote objects, which means that they get passed by-reference over the network. I.e., when used as arguments to a remote call, RMI remote objects do not get copied. A remote reference is created instead and can be used to call methods of the remote object.

Local-only classes are an optimization that allows those clients that are co-located on the same JVM with a given mobile object to access it without the overhead of remote registration. (We discuss the local-only optimization in Section 4.8.1—for now it can be safely ignored.) The implementation classes implement a generated interface that defines all the methods of the original class and extends `java.rmi.Remote`. Remote execution is accomplished by generating an RMI stub for the remote implementation class. We show below a simplified version of the code generated for a class.

```
//Original mobile class A
class A {
 void foo () { ... }
}

//Proxy for A (generated in source code form)
class A implements java.io.Externalizable {
 //ref at different points can point either to
 //local-only or remote implementations, or RMI stub.
 A__interface ref;
 ...
 void foo () {
  try {
   ref.foo ();
  } catch (RemoteException e) {
    //let the user provide custom failure handling
  }
 }//foo
}//A
```

```
//Interface for A (generated in source code form)
interface A__interface extends java.rmi.Remote {
 void foo () throws RemoteException;
}

//Remote implementation (generated in bytecode form
//by modifying original class A)
class A__remote extends UnicastRemoteObject implements
       A__interface {
 void foo () throws RemoteException {...}
}

//Local-only version is identical to remote
//but does not extend UnicastRemoteObject
```

Proxy classes handle several important tasks. One such task is the management of globally unique identifiers. J-Orchestra maintains an "at most one proxy per site" invariant via the help of such globally unique identifiers. Each proxy maintains a unique identifier that it uses to interact with the J-Orchestra runtime system. All proxies implement `java.io.Externalizable` to take full control of their own serialization. This enables the support for object mobility: at serialization time proxies can move their implementation objects as specified by a given mobility scenario. Note that proxy classes are generated in source code, thus enabling the sophisticated user to supply handling code for remote errors.

For anchored classes, proxies provide similar functionality but do not assume the names of their original classes. Since both modifiable and unmodifiable anchored classes cannot change their superclass (to `UnicastRemoteObject`), a different mechanism is required to enable remote execution. An extra level of indirection is added through special purpose classes called *translators*. Translators implement remote interfaces and their purpose is to make anchored classes look like mobile classes as far as the rest of the J-Orchestra rewrite is concerned. Regular proxies, as well as remote and local-only implementation versions are created for translators, exactly like for mobile classes. The code generator puts

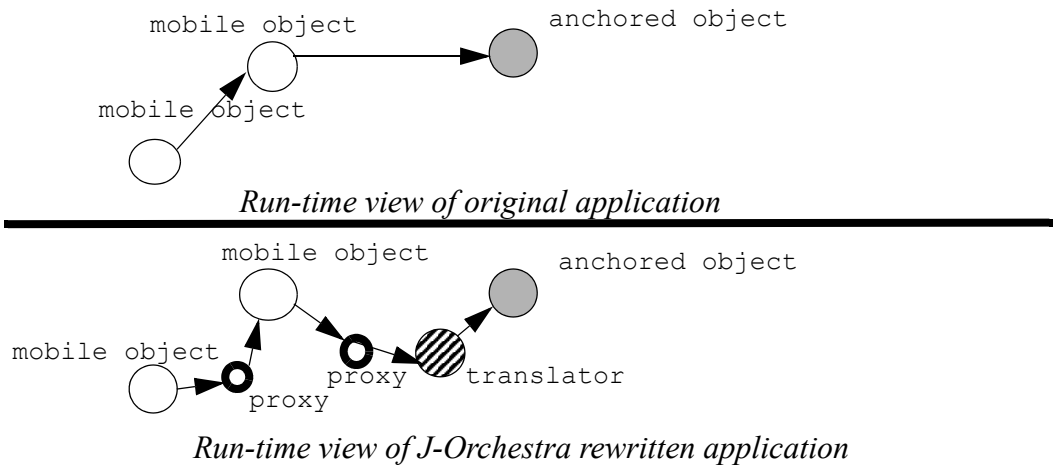


Figure 4-4: Results of the J-Orchestra rewrite schematically. Proxy objects could point to their targets either locally or over the network.

anchored proxies, interfaces and translators into a special package starting with the prefix `remotecapable`. Since it is impossible to add classes to system packages, this approach works uniformly for all anchored classes. Figure 4-4 shows schematically what an object graph looks like during execution of both the original and the J-Orchestra rewritten code. The two levels of indirection introduced by J-Orchestra for anchored classes can be seen. Note that proxies may also refer to their targets indirectly (through RMI stubs) if these targets are on a remote machine.

In addition to giving anchored classes a "remote" identity, translators perform one of the most important functions of the J-Orchestra rewrite: the dynamic translation of direct references into indirect (through proxy) and vice versa, as these references get passed between anchored and mobile code. Consider what happens when references to anchored objects are passed from mobile code (or anchored modifiable code as we will see in the next section) to anchored code. For instance, in Figure 4-5, a mobile application object `o` holds a reference `p` to an object of type `java.awt.Point`. Object `o` can pass reference `p` as an

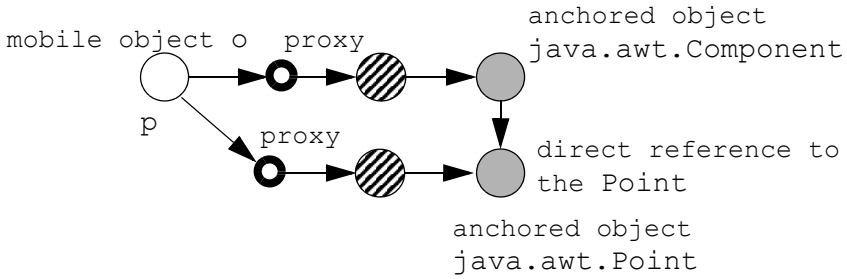


Figure 4-5: Mobile code refers to anchored objects indirectly (through proxies) but anchored code refers to the same objects directly. Each kind of reference should be derivable from the other.

argument to the method `contains` of a `java.awt.Component` object. The problem is that the reference `p` in mobile code is really a reference *to a proxy* for the `java.awt.Point` but the `contains` method cannot be rewritten and, thus, expects a direct reference to a `java.awt.Point` (for instance, so it can assign it or compare it with a different reference). In general, the two kinds of references should be implicitly convertible to each other at run-time, depending on what kind is expected by the code currently being run.

It is worth noting that past systems that follow a similar rewrite as J-Orchestra [31][66][74][76][84] do not offer a translation mechanism. Thus, the partitioned application is safe only if objects passed to unmodifiable (system) code are guaranteed to always be on the same site as that code. This is a big burden to put on the user, especially without analysis tools, like the J-Orchestra classification tool. With the J-Orchestra classification and translation, no object will ever be accessed directly if it can possibly be remote. (See Section 4.9 for some limitations.)

Translation takes place when a method is called on an anchored object. The translator implementation of the method “unwraps” all method parameters (i.e., converts them

from indirect to direct) and "wraps" all results (i.e., converts them from direct to indirect). Since all data exchange between mobile code and anchored code happens through method calls (which go through a translator) we can be certain that references are always of the correct kind. For a code example, consider invoking (from a mobile object) methods `foo` and `bar` in an anchored class `C` passing it a parameter of type `P`. Classes `C` and `P` are packaged in packages `a` and `b`, respectively, and are co-anchored on the same site. The original class `C` and its generated translator are shown below (slightly simplified):

```
//original anchored class C
package a;
class C {
 void foo (b.P p) {...}
 b.P bar () { return new b.P(); }
}

//translator for class C
package remotecapable.a;
class C__translator extends UnicastRemoteObject implements
                C__interface {
 a.C originalC;
 ...
 void foo (remotecapable.b.P p) throws RemoteException {
  originalC.foo ((b.P) Runtime.unwrap(p));
 }

 remotecapable.b.P bar() throws RemoteException {
  return (remotecapable.b.P)Runtime.wrap(originalC.bar());
 }
}
```

4.5.2 Call-Site Wrapping for Anchored Modifiable Code

In the previous section we presented the dynamic conversion of references when calls are made to methods of anchored objects by mobile objects. Nevertheless, wrapping and unwrapping need to also take place when (modifiable) anchored (usually by-choice) objects call other anchored objects that are in a different anchored group. This case is more

complex, but handling it is valuable as it is the only way to enable anchoring by choice. This section explains in detail the wrapping mechanism for anchored modifiable objects.

Anchored and mobile classes present an interesting dichotomy. Anchored objects call methods of all of their co-anchored objects directly without any overhead. Accesses from anchored objects to anchored objects of a different anchored group, on the other hand, result in significant overhead (see Section 4.8) for every method call (because of proxy and translator indirection) and field reference (because direct field references are rewritten to go through method calls). Mobile objects suffer a slightly lower overhead for indirection: calling a method of a mobile object, irrespective of the location of the caller, always results in a single indirection overhead (for the proxy). At the same time, mobile objects can move at will to exploit locality. The result is that if objects of a modifiable class tend to be accessed mostly locally and only rarely remotely, it can be advantageous to anchor this class by choice. In this way, no indirection overhead is incurred for accesses to methods and fields of co-anchored objects. An anchored modifiable class is still remotely accessible (like all classes in a J-Orchestra-rewritten application) but proxies are only used for true remote access.

From a practical standpoint, anchoring by choice is invaluable. It usually allows an application to execute with no slowdown, except for calls that are truly remote. Anchoring by choice is particularly successful when most of the processing in an application occurs on one network site and only some resources (e.g., graphics, sound, keyboard input) are accessed remotely.

Translators of anchored classes, as discussed in the previous section, are the only avenue for data exchange between mobile objects and anchored objects. Translators are a

simple way to perform the wrapping/unwrapping operation because there is no need to analyze and modify the bytecode of the caller: the call is just indirected to go through the translator, which always performs the necessary translations. This approach is sufficient, as long as all the control flow (i.e., the method calls) happens *from* the outside *to* the anchored group but an anchored object never calls methods of objects outside its group. This is the case for pure Java applications consisting of only mobile and anchored unmodifiable (i.e., system) objects. In this case, system code is unaware of application objects and can only call their methods through superclasses or interfaces, in which case no wrapping/unwrapping is required. When anchored modifiable classes are introduced, however, the control-flow patterns become more complex. Anchored modifiable code is regular application code, and thus can call methods in any other application object. Thus, one anchored modifiable object can well be calling an anchored modifiable object in a different anchored group, which may be remote.

Dynamic wrapping/unwrapping needs to take place in this case. The problem is that an anchored modifiable object has direct references to all its co-anchored objects, but may need to pass those direct references to objects outside the anchored group (either mobile or anchored). For instance, imagine a scenario with co-anchored classes `A` and `B`, and class `C`, packaged in packages `a`, `b`, and `c`, respectively, and anchored on a different site. The original application code may look like the following:

```
package a;
class A {

 b.B b;

 c.C c;
```

```
void baz () {
  c.foo (b);
  b.B b = c.bar ();
 }
}

package b;
class B {...}


package c;
class C {
 void foo (b.B b) {...}
 b.B bar () {...}
}
```

If we were to perform a straightforward rewrite of class `A` to refer to `B` directly but to `C` by proxy we would get:

```
package a;
class A {
 b.B b;
 remotecapable.c.C c;
 void baz  () {
  c.foo (b); //incorrectly passing
             //a direct reference to b.B!
  b.B b = c.bar();//incorrectly returning
                  //an indirect ref. to b.B!
 }
}

//proxy for class C
package remotecapable.c;
class C {
 ...
 void foo (remotecapable.b.B b) {...}
 remotecapable.b.B bar () {...}
}
```

As indicated by the comments in the code, this rewrite would result in erroneous bytecodes: direct references are passed to code that expects an indirection and vice versa. A fix could be applied in two places: either at the call site (e.g., the code in class `A` that calls

`c.bar()`) or at the indirection site (i.e., at the proxy `C`, or at some other intermediate object, analogous to the translators we saw in the previous section). The translators of the previous section do the wrapping/unwrapping at the indirection site. Unfortunately this solution is not applicable here. If we were to do the wrapping/unwrapping inside the proxy, the proxy for `C` would look like:

```
// This is imaginary code!
//Irrelevant details (e.g., exception handling) omitted
package remotecapable.c;
class C {
 C__interface ref;
 ...


 // used when caller is outside B's anchored group
 void foo (remotecapable.b.B b) {
  ref.foo ((b.B) Runtime.unwrap(b));
 }
 // used when caller is in B's anchored group
 void foo (b.B b) {
  ref.foo((remotecapable.b.B) Runtime.wrap(b));
 }
 // used when caller is outside B's anchored group
 remotecapable.b.B bar() {
  return ref.bar();
 }
 // used when caller is in B's anchored group
 b.B bar() {
  return ((b.B) Runtime.unwrap(ref.bar());
 }
}
```

Unfortunately, the last two methods differ only in their return type, thus overloading cannot be used to resolve a call to `bar`. This is why a call-site rewrite is required. Since J-Orchestra operates at the bytecode level, this action is not trivial. We need to analyze the bytecode, reconstruct argument types, see if a conversion is necessary, and insert code to

wrap and unwrap objects. The resulting code for our example class `A` is shown below (in source code form, for ease of exposition).

```
package a;
class A {
 b.B b;
 remotecapable.c.C c;


 void baz  () {
  //wrap b in the call to foo
  c.foo ((remotecapable.b.B)Runtime.wrap (b));
  //unwrap b after the call to bar
  b.B b = (b.B) Runtime.unwrap (c.bar());
 }
}
```

A special case of the above problem is self-reference. An object always refers to itself (`this`) directly. If it attempts to pass such references outside its anchored group (or, in the case of a mobile object, to any other object) the reference should be wrapped.

4.5.3 Placement Policy Based On Creation Site

The class-based distribution of J-Orchestra is powerful and useful enough for most application scenarios. Using a class as a distribution unit makes assigning classes (or groups of classes) to their destination network sites manageable even for medium to large applications. However, sometimes a distribution policy that is more fine-grained than class-level can become necessary. For example, a meaningful distribution might require placing different instances of the same class on different network sites.

The J-Orchestra creation site placement policy provides an approach that enables such placement. This advanced feature allows the user to distinguish between different instances of the same class, based on the points in the program at which these instances are

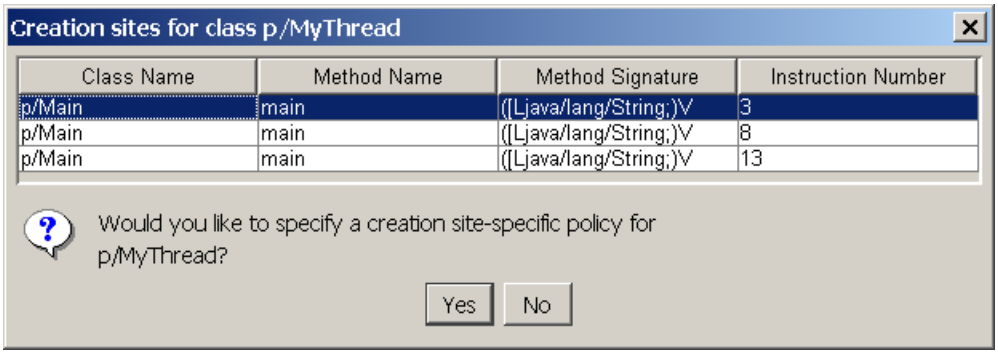


Figure 4-6: The results of a query on the creation sites of class p.MyThread.

instantiated. What complicates the implementation of this feature is that because J-Orchestra operates at the bytecode level, source code can not be used to identify such instantiation points.

To demonstrate the creation site placement policy, let us consider the following example. Class `p.MyThread` extends a systems class `java.lang.Thread`, and class `p.Main` has a method `main` that instantiates and calls method `start` on three instances of `p.MyThread` as follows:

```
public static void main (String args[]) {


 p.MyThread thread1 = new p.MyThread ("Thread #1");
 p.MyThread thread2 = new p.MyThread ("Thread #2");
 p.MyThread thread3 = new p.MyThread ("Thread #3");


 thread1.start();
 thread2.start();
 thread3.start();

}
```

Under the standard J-Orchestra partitioning, the classification heuristic would classify class `p.MyThread` as anchored (i.e., it controls threading, a native platform-specific resource), and all its instances would have to be created on the same network site. However, the user can override the classification results by using the creation site specific placement policy.[4] To accomplish that, the user first inquires about the points in the code at which the instances of `p.MyThread` are instantiated. Figure 4-6 shows a GUI dialog box through which the system displays the requested information. As one can see in the dialog box, the system uniquely identifies a creation site by listing its locations, each of which consisting of a class, a method, a method signature, and an instruction number. While the first three parts of a location are self explanatory, the instruction number is simply a heuristic. Because J-Orchestra operates at the bytecode level, the system uses the index of the constructor call instruction in the sequence of the method's bytecodes. After the user chooses a creation site specific policy for a particular class, all creation locations of a class become separate distribution entities that are added to the main tree view of the J-Orchestra. The user can subsequently assign these new distribution entities to separate destination network sites.

4.5.4 Object Mobility

One of the ways in which the advanced J-Orchestra user can tune partitioned applications to improve distributed performance is through the use of mobility policies. Object mobility can significantly affect the performance of a distributed application. Mobile objects can exploit application locality and eliminate the need for network communication.

4. Such overriding is done at the user's own risk, for thread objects that might be passed to native code will no longer be "anchored" on the same site.

Apart from the creation site placement policy, mobility is the only other mechanism in J-Orchestra that enables per-instance instead of per-class treatment. That is, two objects of the same mobile class can behave entirely differently at run-time based on their uses (i.e., to which methods they are passed as parameters, etc.). Object mobility in J-Orchestra is synchronous: objects move in response to method calls. J-Orchestra supports three object moving scenarios: moving a parameter of a remote method call to the site of the call, moving the return value of a remote method call to the site of the caller, and moving "this" object to the site of the call. In terms of design, our object migration policies are similar to what is commonly found in the mobile objects literature [11][39]. In terms of mechanisms, our implementation bears many similarities to the one in JavaParty [31].

Specifically, J-Orchestra supports mobility through a programming interface and runtime services. Recall that J-Orchestra proxies are generated in source code form. This makes it fairly straightforward to generate additional mobility-specific methods in mobile classes proxies. The user can then use these generated methods as primitives for specifying various mobility scenarios. In addition, each mobile proxy contains a data member of type `MigrationSchema`, which specifies how the object pointed to by the proxy should move. The default value of `MigrationSchema` is `by-reference`, which means that an RMI stub is sent whenever a proxy is passed as a parameter or returned as a result of a remote method call. Mobile proxies enable flexible migration policies by implementing their own serialization. Assigning the value `by-move` to the `MigrationSchema` of a mobile proxy will have the object to which it is pointing move to a remote site. The following generated methods in mobile proxies can be used to specify mobility policies for moving a parameter

of a remote method call to the site of the call and moving the return value of a remote method call to the site of the caller.

```
private MigrationSchema _migrationSchema;
public void setMigrationSchema (MigrationSchema schema)
{...}

public MigrationSchema getMigrationSchema () { ... }


//Overwrite standard serialization behavior
public void writeExternal (ObjectOutput out)
                                    throws IOException {

 Marshaller.marshall(out, this);

}

public void readExternal (ObjectInput in)
           throws IOException, ClassNotFoundException {

 Marshaller.unmarshall(in, this);

}
```

The code below is a (slightly simplified) example of specifying that the parameter `p` of the remote method `foo` should move when the remote method invocation takes place.

```
//proxy method; P is a proxy of a mobile class
public void foo (P p) {
 try {
  //the object pointed by p will move to the site
  //of the method foo, unless p and foo are
  //already collocated.
  p.getMigrationSchema().setByMove();
  //the migration will take place during
  //the serialization of p as part of
  //the invocation of foo.
  _ref.foo (p);

 } catch (RemoteException e) {...}
}
```

The J-Orchestra mobility API contains the following two methods, which can be used to move "this" object (i.e., the one pointed to by the mobile proxy) to and from the site of a remote method invocation.

```
public void moveToRemoteSite (ObjectFactory remoteFac)
{...}

public void moveFromRemoteSite (ObjectFactory remoteFac)
{...}
```

The code below demonstrates how the user can modify the proxy to specify that "this" object should temporarily move over to the local machine to invoke method `bar` locally.

```
//proxy method
public void bar () {
 try {
  ObjectFactory remoteObjectFactory =
             getObjectFactory("SomeSymbolicFactoryName");

  //moves _ref from the remote site, identified by
  //remoteObjectFactory, to the local machine
  moveFromRemoteSite(remoteObjectFactory);
  //execute the call locally
  _ref.bar();
  //moves _ref back to the remote site
  moveToRemoteSite(remoteObjectFactory);

 } catch (RemoteException e) {...}
}
```

One element of the runtime support for mobility in J-Orchestra is the `Marshaller` class, which enables mobility at serialization time. Another important piece of the runtime functionality preserves the "at most one proxy per site" invariant. Because proxies contain unique identifiers, when unserializing a proxy at a remote site, the runtime service checks

whether a proxy with the same unique identifier already exists; if the answer is affirmative, the existing proxy is used instead of instantiating a new one.

An object that is being moved might contain some embedded proxies to other objects, transitively reachable from it. This presents some interesting opportunities for specifying complex mobility scenarios. For example, if object `P` moves, move also objects `Q` and `R`, if they are transitively reachable from it. The existing J-Orchestra infrastructure can be easily extended to support such mobility scenarios, and we would like to pursue this as a possible future work direction.

4.6 Dealing with Concurrency and Synchronization

One of the primary design goals of J-Orchestra is to be able to run partitioned programs with standard Java middleware. However, Java middleware mechanisms, such as Java RMI or CORBA implementations, do not support thread coordination over the network: synchronizing on remote objects does not work correctly, and thread identity is not preserved for executions spanning multiple machines. Prior approaches to dealing with the problem suffer from one of two weaknesses: either they require a new middleware mechanism, or they add overhead to the execution to propagate a thread identifier through all method calls. Therefore, these weaknesses leave the existing approaches unable to meet the design goals of J-Orchestra, necessitating a new approach that should work with an unmodified middleware implementation efficiently. We next describe the design, implementation, and evaluation of the J-Orchestra approach to this problem.

4.6.1 Overview and Existing Approaches

J-Orchestra enables Java thread synchronization in a distributed setting. This mechanism addresses monitor-style synchronization (mutexes and condition variables), which is well-suited for a distributed threads model. (This is in contrast to low-level Java synchronization, such as volatile variables and atomic operations, which are better suited for symmetric multiprocessor machines.)

This solution is not the first in this design space. Past solutions fall in two different camps. A representative of the first camp is the approach of Haumacher et al. [30], which proposes a replacement of Java RMI that maintains correct multithreaded execution over the network. If employing special-purpose middleware is acceptable, this approach is sufficient. Nevertheless, it would not be suitable for J-Orchestra, which has the ability to use standard middleware as one of its primary design objectives. In general, it is often not desirable to move away from standard middleware, for reasons of portability and ease of deployment. Therefore, the second camp, represented by the work of Weyns, Truyen, and Verbaeten [98], advocates transforming the client application instead of replacing the middleware. Unfortunately, clients (i.e., callers) of a method do not know whether its implementation is local or remote. Thus, to support thread identity over the network, *all* method calls in an application need to be automatically re-written to pass one extra parameter—the thread identifier. This imposes both space and time overhead: extra code is needed to propagate thread identifiers, and adding an extra argument to every call incurs a run-time cost. Weyns, Truyen, and Verbaeten [98] quantify this cost to about 3% of the total execution time of an application. Using more representative macro-benchmarks (from the SPEC JVM suite) we found the cost to be between 5.5 and 12% of the total execution time. A secondary

disadvantage of the approach is that the transformation becomes complex when application functionality can be called by native system code, as in the case of application classes implementing a Java system interface.

J-Orchestra implements a technique that addresses both the problem of portability and the problem of performance. This technique follows the main lines of the approach of Weyns, Truyen, and Verbaeten: it replaces all monitor operations in the bytecode (such as `monitorenter`, `monitorexit`, `Object.wait`) with calls to operations of J-Orchestra distribution-aware synchronization library. Nevertheless, it avoids instrumenting every method call with an extra argument. Instead, it performs a bytecode transformation on the generated RMI stubs. The transformation is general and portable: almost every RPC-style middleware mechanism needs to generate stubs for the remotely invokable methods. By transforming those when needed, it can propagate thread identity information for all remote invocations, without unnecessarily burdening local invocations. This approach also has the advantage of simplicity with respect to native system code. Finally, the J-Orchestra implementation of the approach is fine-tuned, making the total overhead of synchronization be negligible (below 4% overhead even for empty methods and no network cost).

4.6.2 Distributed Synchronization Complications

Modern mainstream languages such as Java or C# have built-in support for concurrency. Specifically, Java provides the class `java.lang.Thread` for creating and managing concurrency, monitor methods `Object.wait`, `Object.notify`, and `Object.notifyAll` for managing state dependence, and `synchronized` methods and code blocks for maintaining exclusion among multiple concurrent activities. (An excellent reference for multithreading in Java is Lea's textbook [47].)

Concurrency constructs usually do not interact correctly with middleware implementations, however. In particular, Java RMI does not propagate synchronization operations to remote objects and does not maintain thread identity across different machines.

To see the first problem, consider a Java object `obj` that implements a `Remote` interface `RI` (i.e., a Java interface `RI` that extends `java.rmi.Remote`). Such an object is remotely accessible through the `RI` interface. That is, if a client holds an interface reference `r_ri` that points to `obj`, then the client can call methods on `obj`, even though it is located on a different machine. The implementation of such remote access is the standard RPC middleware technique: the client is really holding an indirect reference to `obj`. Reference `r_ri` points to a local RMI "stub" object on the client machine. The stub serves as an intermediary and is responsible for propagating method calls to the `obj` object. What happens when a monitor operation is called on the remote object, however? There are two distinct cases: Java calls monitor operations (locking and unlocking a mutex) implicitly when a method labeled `synchronized` is invoked and when it returns. This case is handled correctly through RMI, since the stub will propagate the call of a synchronized remote method to the correct site. Nevertheless, all other monitor operations are not handled correctly by RMI. For instance, a `synchronized` block of code in Java corresponds to an explicit mutex lock operation. The mutex can be the one associated with any Java object. Thus, when clients try to explicitly synchronize on a remote object, they end up synchronizing on its stub object instead. This does not allow threads on different machines to synchronize using remote objects: one thread could be blocked or waiting on the real object `obj`, while the other thread may be trying to synchronize on the stub instead of on the `obj` object. Similar problems exist for all other monitor operations. For instance, RMI cannot be used to

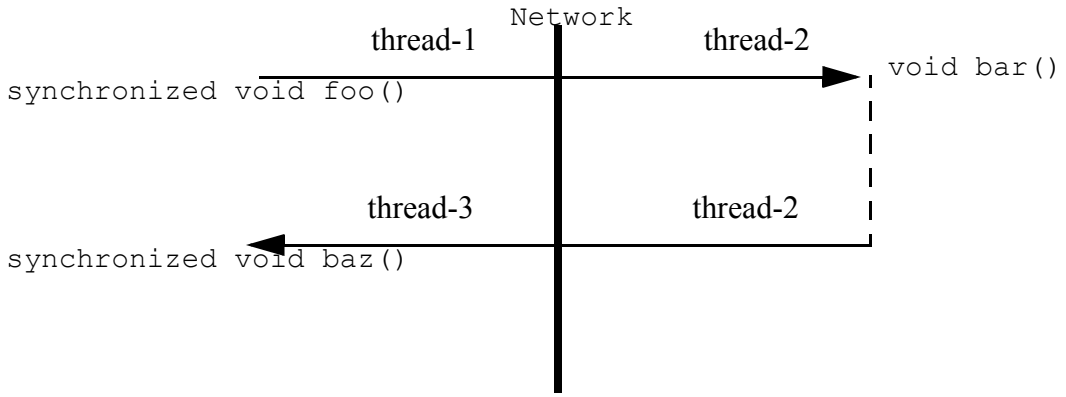


Figure 4-7: The zigzag deadlock problem in Java RMI.

propagate monitor operations such as `Object.wait`, `Object.notify`, over the network. The reason is that these operations cannot be indirected: they are declared in class `Object` to be `final`, which means that the methods can not be overridden in subclasses that implement the `Remote` interfaces required by RMI.

The second problem concerns preserving thread identities in remote calls. The Java RMI runtime starts a new thread for each incoming remote call. Thus, a thread performing a remote call has no memory of its identity in the system. Figure 4-7 demonstrates the so-called "zigzag deadlock problem", common in distributed synchronization. Conceptually, methods `foo`, `bar`, and `baz` are all executed in the same thread—but the location of method `bar` happens to be on a remote machine. In actual RMI execution, thread-1 will block until `bar`'s remote invocation completes, and the RMI runtime will start a new thread for the remote invocations of `bar` and `baz`. Nevertheless, when `baz` is called, the monitor associated with thread-1 denies entry to thread-3: the system does not recognize that thread-3 is just handling the control flow of thread-1 after it has gone through a remote machine. If no special care is taken, a deadlock condition occurs.

4.6.3 Solution: Distribution-Aware Synchronization

As we saw, any solution for preserving the centralized concurrency and synchronization semantics in a distributed environment must deal with two issues: each remote method call can be executed on a new thread, and standard monitor methods such as `Object.wait`, `Object.notify`, and `synchronized` blocks can become invalid when distribution takes place. Taking these issues into account, we maintain per-site "thread id equivalence classes," which are updated as execution crosses the network boundary; and at the bytecode level, we replace all the standard synchronization constructs with the corresponding method calls to a per-site synchronization library. This synchronization library emulates the behavior of the monitor methods, such as `monitorenter`, `monitorexit`, `Object.wait`, `Object.notify`, and `Object.notifyAll`, by using the thread id equivalence classes. Furthermore, these synchronization library methods, unlike the `final` methods in class `Object` that they replace, get correctly propagated over the network using RMI when necessary so that they execute on the network site of the object associated with the monitor.

In more detail, our approach consists of the following steps:

- Every instance of a monitor operation in the bytecode of the application is replaced, using bytecode rewriting, by a call to our own synchronization library, which emulates the monitor-style synchronization primitives of Java
- Our library operations check whether the target of the monitor operation is a local object or an RMI stub. In the former case, the library calls its local monitor operation. In the latter case, an RMI call to a remote site is used to invoke the appropriate library operation on that site. This solves the problem of propagating monitor operations over the network. We also apply a compile-time optimization to this step: using a simple static

analysis, we determine whether the target of the monitor operation is an object that is known statically to be on the current site. This is the case for monitor operations on the `this` reference, as well as other objects of anchored types that J-Orchestra guarantees will be on the same site throughout the execution. If we know statically that the object is local, we avoid the runtime test and instead call a local synchronization operation.

- Every remote RMI call, whether on a synchronized method or not, is extended to include an extra parameter. The instrumentation of remote calls is done by bytecode transformation of the RMI stub classes. The extra parameter holds the thread equivalence class for the current calling thread. Our library operations emulate the Java synchronization primitives but do not use the current, machine-specific thread id to identify a thread. Instead, a mapping is kept between threads and their equivalence classes and two threads are considered the same if they map to the same equivalence class. Since an equivalence class can be represented by any of its members, our current representation of equivalence classes is compact: we keep a combination of the first thread id to join the equivalence class and an id for the machine where this thread runs. This approach solves the problem of maintaining thread identity over the network.

We illustrate the above steps with examples that show how they solve each of the two problems identified earlier. We first examine the problem of propagating monitor operations over the network. Consider a method as follows:

```
//original code
void foo (Object some_remote_object) {

 this.wait();
 ...
 some_remote_object.notify();
 ...


}
```

At the bytecode level, method `foo` will have a body that looks like:

```
//bytecode
aload_0
invokevirtual     java.lang.Object.wait
...
aload_1
invokevirtual     java.lang.Object.notify
...
```

Our rewrite will statically detect that the first monitor operation (`wait`) is local, as it is called on the current object itself (`this`). The second monitor operation, however, is (potentially) remote and needs to be redirected to its target machine using an RMI call. The result is shown below:

```
//rewritten bytecode
aload_0
//dispatched locally
invokestatic jorchestra.runtime.distthreads.wait_
...
aload_1
//get thread equivalence info from runtime
invokestatic
jorchestra.runtime.ThreadInfo.getThreadEqClass
//dispatched through RMI;
//all remote interfaces extend DistSyncSupporter
invokeinterface jorchestra.lang.DistSynchSupporter.notify_
...
```

(The last instruction is an interface call, which implies that each remote object needs to support monitor methods, such as `notify_`. This may seem to result in code bloat at first, but our transformation adds these methods to the topmost class of each inheritance hierarchy in an application, thus minimizing the space overhead.)

Let's now consider the second problem: maintaining thread identity over the network. Figure 4-8 demonstrates how using the thread id equivalence classes can solve the "zigzag deadlock problem" presented above. These thread id equivalence classes enable

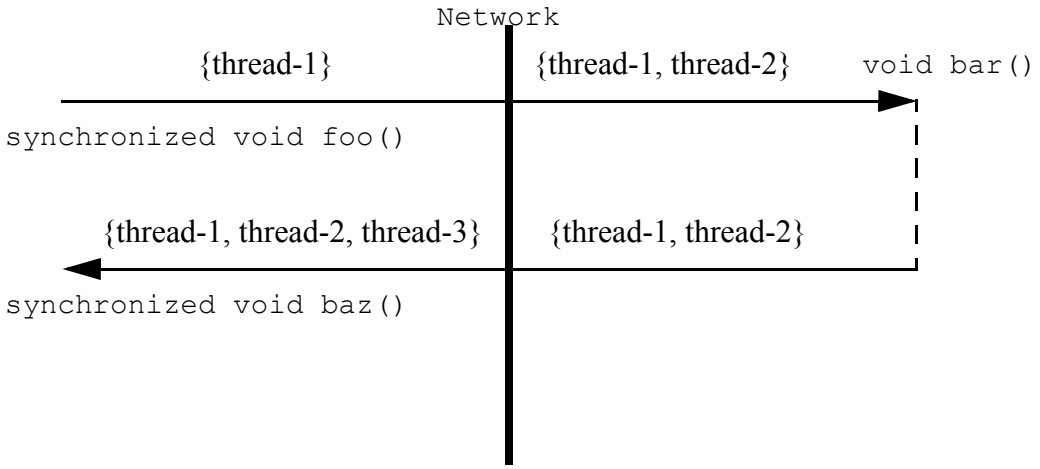


Figure 4-8: Using thread id equivalence classes to solve the "zigzag deadlock problem" in Java RMI.

our custom monitor operations to treat all threads within the same equivalence class as the same thread. (We illustrate the equivalence class by listing all its members in the figure, but, as mentioned earlier, in the actual implementation only a single token that identifies the equivalence class is passed across the network.) More specifically, our synchronization library is currently implemented using regular Java mutexes and condition variables. For instance, the following code segment (slightly simplified) shows how the library emulates the behavior of the bytecode instruction `monitorenter`. (For readers familiar with monitor-style concurrent programming, our implementation should look straightforward.) The functionality is split into two methods: the static method `monitorenter` finds or creates the corresponding `Monitor` object associated with a given object: our library keeps its own mapping between objects and their monitors. The member method `enter` of class `Monitor` causes threads that are not in the equivalence class of the holder thread to wait until the monitor is unlocked.

```
public static void monitorenter (Object o) {
 Monitor this_m = null;
 synchronized (Monitor.class) {
  this_m = (Monitor)_objectToMonitor.get(o);
  if (this_m == null) {
   this_m = new Monitor();
   _objectToMonitor.put(o, this_m);
  }
 } //synchronized
 this_m.enter();
}


private synchronized void enter () {
 while (_timesLocked != 0 &&
        curThreadEqClass != _holderThreadId)
  try { wait(); } catch(InterruptedException e) {...}

  if (_timesLocked == 0) {
  _holderThreadId = getThreadID();
 }
 _timesLocked++;
}
```

The complexity of maintaining thread equivalence classes determines the overall efficiency of the solution. The key to efficiency is to update the thread equivalence classes only when necessary—that is, when the execution of a program crosses the network boundary. Adding the logic for updating equivalence classes at the beginning of every remote method is not the appropriate solution: in many instances, remote methods can be invoked locally within the same JVM. In these cases, adding any additional code for maintaining equivalence classes to the remote methods themselves would be unnecessary and detrimental to performance. In contrast, our solution is based on the following observation: the program execution will cross the network boundary only after it enters a method in an RMI stub. Thus, RMI stubs are the best location for updating the thread id equivalence classes on the client site of a remote call.

Adding custom logic to RMI stubs can be done by modifying the RMI compiler, but this would negate our goal of portability. Therefore, we use bytecode engineering on standard RMI stubs to retrofit their bytecode so that they include the logic for updating the thread id equivalence classes. This is done completely transparently relative to the RMI runtime by adding special delegate methods that look like regular remote methods, as shown in the following code example. To ensure maximum efficiency, we pack the thread equivalence class representation into a long integer, in which the less significant and the most significant 4 bytes store the first thread id to join the equivalence class and the machine where this thread runs, respectively. This compact representation significantly reduces the overhead imposed on the remote method calls, as we demonstrate later on. Although all the changes are applied to the bytecode directly, we use source code for ease of exposition.

```
//Original RMI stub: two remote methods foo and bar
class A_Stub ... {
 ...
 public void foo (int i) throws RemoteException {...}
 public int bar () throws RemoteException {...}
}

//Retrofitted RMI stub
class A_Stub ... {
 ...
 public void foo (int i) throws RemoteException {
   foo__tec (Runtime.getThreadEqClass(), i);
 }

 public void foo__tec (long tec, int i)
                                    throws RemoteException
 {...}

 public int bar () throws RemoteException {
   return bar__tec (Runtime.getThreadEqClass());
 }
```

```
 public int bar__tec (long tec) throws RemoteException
 {...}
}
```

Remote classes on the callee site provide symmetrical delegate methods that update the thread id equivalence classes information according to the received `long` parameter, prior to calling the actual methods. Therefore, having two different versions for each remote method (with the delegate method calling the actual one) makes the change transparent to the rest of the application: neither the caller of a remote method nor its implementor need to be aware of the extra parameter. Remote methods can still be invoked directly (i.e., not through RMI but from code on the same network site) and in this case they do not incur any overhead associated with maintaining the thread equivalence information.

4.6.4 Benefits of the Approach

The two main existing approaches to the problem of maintaining the centralized Java concurrency and synchronization semantics in a distributed environment have involved either using custom middleware [30] or making universal changes to the distributed program [98]. We argue next that our technique is a good fit for J-Orchestra, being more portable than using custom middleware and more efficient than a universal rewrite of the distributed program. Finally, we quantify the overhead of our approach and show that our implementation is indeed very efficient.

4.6.4.1 Portability

In our context, a solution for preserving the centralized concurrency and synchronization semantics in a distributed environment is only as useful as it is portable. A solution is portable if it applies to different versions of the same middleware (e.g., past and future

versions of Java RMI) and to different middleware mechanisms such as CORBA and .NET Remoting. Our approach is both simple and portable to other middleware mechanisms, because it is completely orthogonal to other middleware functionality: We rely on bytecode engineering, which allows transformations without source code access, and on adding a small set of runtime classes to each network node of a distributed application. The key to our transformation is the existence of a client stub that redirects local calls to a remote site. Using client stubs is an almost universal technique in modern middleware mechanisms. Even in the case when these stubs are generated dynamically, our technique is applicable, as long as it is employed at class load time.

For example, our bytecode instrumentation can operate on CORBA stubs as well as it does on RMI ones. Our stub transformations simply consist of adding delegate methods (one for each client-accessible remote method) that take an extra thread equivalence parameter. Thus, no matter how complex the logic of the stub methods is, we would apply to them the same simple set of transformations.

Some middleware mechanisms such as the first version of Java RMI also use server-side stubs (a.k.a. *skeletons*) that dispatch the actual methods. Instead of presenting complications, skeletons would even make our approach easier. The skeleton methods are perfect for performing our server-side transformations, as we can take advantage of the fact that the program execution has certainly crossed the network boundary if it entered a method in a skeleton. Furthermore, having skeletons to operate on would eliminate the need to change the bytecodes of the remote classes. Finally, the same argument of the simplicity of our stub transformations being independent of the complexity of the stub code itself equally applies to the skeleton transformations.

In a sense, our approach can be seen as adding an orthogonal piece of functionality (concurrency control) to existing distribution middleware. In this sense, one can argue that the technique has an aspect-oriented flavor.

4.6.4.2 The Cost of Universal Extra Arguments

Our approach eliminates both the runtime and the complexity overheads of the closest past techniques in the literature. Weyns, Truyen, and Verbaeten [98][99] have advocated the use of a bytecode transformation approach to correctly maintain thread identity over the network. Their technique is occasionally criticized as "incur[ring] great runtime overhead" [30]. The reason is that, since clients do not know whether a method they call is local or remote, every method in the application is extended with an extra argument—the current thread id—that it needs to propagate to its callees. Weyns et al. argue that the overhead is acceptable and present limited measurements where the overhead of maintaining distributed thread identity is around 3% of the total execution time. Below we present more representative measurements that put this cost at between 5.5 and 12%. A second cost that has not been evaluated, however, is that of complexity: adding an extra parameter to all method calls is hard when some clients cannot be modified because, e.g., they are in native code form or access the method through reflection. In these cases a correct application of the Weyns et al. transformation would incur a lot of complexity. This complexity is eliminated with our approach.

It is clear that some run-time overhead will be incurred if an extra argument is added and propagated to every method in an application. To see the range of overhead, we wrote a simple micro-benchmark, in which each method call performs one integer arithmetic operation, two comparisons and two (recursive) calls. Then we measured the overhead of

adding one extra parameter to each method call. Figure 4-1 shows the results of this benchmark. For methods with 1-5 integer arguments we measure their execution time with one extra reference argument propagated in all calls. As seen, the overhead varies unpredictably but ranges from 5.9 to 12.7%.

Table 4-1. Micro-benchmark: overhead of method calls with one extra argument.

#params	1 (base)	1+1	2+1	3+1	4+1	5+1
Execution time (sec) for 10^8 calls	1.945	2.059	2.238	2.523	2.691	2.916
Slowdown relative to previous	-	5.9%	8.7%	12.7%	6.7%	8.4%

Nevertheless, it is hard to get a representative view of this overhead from micro-benchmarks, especially when running under a just-in-time compilation model. Therefore, we concentrated on measuring the cost on realistic applications. As our macro-benchmarks, we used applications from the SPEC JVM benchmark suite. Of course, some of the applications we measured may not be multithreaded, but their method calling patterns should be representative of multithreaded applications, as well.

We used bytecode instrumentation to add an extra reference argument to all methods and measured the overhead of passing this extra parameter. In the process of instrumenting realistic applications, we discovered the complexity problems outlined earlier. The task of adding an extra parameter is only possible when all clients can be modified by the transformation. Nevertheless, all realistic Java applications present examples where clients will not be modifiable. An application class can be implementing a system interface, making native Java system code a potential client of the class's methods. For instance, using class frameworks, such as AWT, Swing, or Applets, entails extending the classes provided by such frameworks and overriding some methods with the goal of customizing the application's

behavior. Consider, for example, a system interface `java.awt.TextListener`, which has a single method `void textValueChanged (TextEvent e)`. A non-abstract application class extending this interface has to provide an implementation of this method. It is impossible to add an extra parameter to the method `textValueChanged` since it would prevent the class from being used with AWT. Similarly a Java applet overrides methods `init`, `start`, and `stop` that are called by Web browsers hosting the applet. Adding an extra argument to these methods in an applet would invalidate it. These issues can be addressed by careful analysis of the application and potentially maintaining two interfaces (one original, one extended with an extra parameter). Nevertheless, this would result in code bloat, which could further hinder performance.

Since we were only interested in quantifying the potential overhead of adding and maintaining an extra method parameter, we sidestepped the complexity problems by avoiding the extra parameter for methods that could be potentially called by native code clients. Instead of changing the signatures of such methods so that they would take an extra parameter, we created the extra argument as a local variable that was passed to all the callees of the method. The local variable is never initialized to a useful value, so no artificial overhead is added by this approach. This means that our measurements are slightly conservative: we do not really measure the cost of correctly maintaining an extra thread identity argument but instead conservatively estimate the cost of passing one extra reference parameter around. Maintaining the correct value of this reference parameter, however, may require some extra code or interface duplication, which may make performance slightly worse than what we measured.

Another complication concerns the use of Java reflection for invoking methods, which makes adding an extra argument to such methods impossible. In fact, we could not correctly instrument all the applications in the SPEC JVM suite, exactly because some of them use reflection heavily and we would need to modify such uses by hand.

The results of our measurements appear in Table 4-2. The table shows total execution time for four benchmarks (compress—a compression utility, javac—the Java compiler, mtrt—a multithreaded ray-tracer, and jess—an expert system) in both the original and instrumented versions, as well as the slowdown expressed as the percentage of the differences between the two versions, ranging between 5.5 and 12%. The measurements were on a 600MHz Pentium III laptop, running JDK 1.4.

Table 4-2. Macro-benchmarks: cost of a universal extra argument.

Benchmark	compress	javac	mtrt	jess
Original version (sec)	22.403	19.74	6.82	8.55
Instrumented version (sec)	23.644	21.18	7.49	9.58
Slowdown	5.54%	7.31%	9.85%	12.05%

The best way to interpret these results is as the overhead of pure computation (without communication) that these programs would incur under the Weyns et al. technique if they were to be partitioned with J-Orchestra so that their parts would run correctly on distinct machines. We see, for instance, that running jess over a network would incur an overhead of 12% in extra computation, just to ensure the correctness of the execution under multiple threads. Our approach eliminates this overhead completely: overhead is only incurred when actual communication over distinct address spaces takes place. As we show next, this overhead is minuscule relative to total execution time, even for an infinitely fast network and no computation performed by remote methods.

4.6.4.3 Maintaining Thread Equivalence Classes Is Cheap

Maintaining thread equivalence classes, which consists of obtaining, propagating, and updating them, constitutes the overhead of our approach. In other words, to maintain the thread equivalence classes correctly, each retrofitted remote method invocation includes one extra local method call on the client side to obtain the current class, an extra argument to propagate it over the network, and another local method call on the server side to update it. The two extra local calls, which obtain and update thread equivalence classes, incur virtually no overhead, having a hash table lookup as their most expensive operation and causing no network communication. Thus, the cost of propagating the thread equivalence class as an extra argument in each remote method call constitutes the bulk of our overhead.

In order to minimize this overhead, we experimented with different thread equivalence classes' representations. We performed preliminary experiments which showed that the representation does matter: the cost of passing an extra reference argument (any subclass of `java.lang.Object` in Java) over RMI can be high, resulting in as much as 50% slowdown in the worst case. This happens because RMI accomplishes the marshalling/unmarshalling of reference parameters via Java serialization, which involves dynamic memory allocation and the use of reflection. Such measurements led us to implement the packed representation of thread equivalence class information into a long integer, as described earlier. A `long` is a primitive type in Java, hence the additional cost of passing one over the network became negligible.

To quantify the overall worst-case overhead of our approach, we ran several microbenchmarks, measuring total execution time taken by invoking empty remote meth-

ods with zero, one `java.lang.String`, and two `java.lang.String` parameters. Each remote method call was performed 10^6 times. The base line shows the numbers for regular uninstrumented RMI calls. To measure the pure overhead of our approach, we used an unrealistic setting of collocating the client and the server on the same machine, thus eliminating all the costs of network communication. The measurements were on a 2386MHz Pentium IV, running JDK 1.4. The results of our measurements appear in Table 4-3.

Table 4-3. Overhead of Maintaining Thread Equivalence Classes

No. of Params	Base Line (ms)	Maintaining Thread Equivalence Classes (ms)	Overhead (%)
0	145,328	150,937	3.86%
1	164,141	166,219	1.27%
2	167,984	168,844	0.51%

Since the remote methods in this benchmark did not perform any operations, the numbers show the time spent exclusively on invoking the methods. While the overhead is approaching 4% for the remote method without any parameters, it diminishes gradually to half a percent for the method taking two parameters. Of course, our settings for this benchmark are strictly worst-case—had the client and the server been separated by a network or had the remote methods performed any operations, the overhead would strictly decrease.

4.6.5 Discussion

As we mentioned briefly earlier, the J-Orchestra distributed synchronization approach only supports monitor-style concurrency control. This is a standard application-level concurrency control facility in Java, but it is not the only one and the language has currently evolved to better support other models. For example, high-performance applications may use `volatile` variables instead of explicit locking. In fact, use of non-monitor-

style synchronization in Java will probably become more popular in the future. The JSR-166 specification has standardized many concurrent data structures and atomic operations in Java 5. Although our technique does not support all the tools for managing concurrency in the Java language, this is not so much a shortcoming as it is a reasonable design choice. Low-level concurrency mechanisms (volatile variables and their derivatives) are useful for synchronization in a single memory space. Their purpose is to achieve optimized performance for symmetric multiprocessor machines. In contrast, our approach deals with correct synchronization over middleware—i.e., it explicitly addresses distributed memory, resulting from partitioning. Programs partitioned with J-Orchestra are likely to be deployed on a cluster or even a more loosely coupled network of machines. In this setting, monitor-style synchronization makes perfect sense.

On the other hand, in the future we can use the lower-level Java concurrency control mechanisms to optimize the J-Orchestra runtime synchronization library for emulating Java monitors. As we saw in Section 4.6.3, our current library is itself implemented using monitor-style programming (`synchronized` blocks, `Object.wait`, etc.). With the use of optimized low-level implementation techniques, we can gain in efficiency. We believe it is unlikely, however, that such a low-level optimization in our library primitives will make a difference for most client applications of our distributed synchronization approach.

Finally, we should mention that our current implementation does not handle all the nuances of Java monitor-style synchronization, but the issue is one of straightforward engineering. Notably, we do not currently propagate `Thread.interrupt` calls to all the nodes that might have threads blocked in an invocation of the `wait` method. Even though it is unlikely that the programs amenable to automatic partitioning would ever use the `inter-`

`rupt` functionality, our design can easily support it. We can replace all the calls to `Thread.interrupt` with calls to our synchronization library, which will obtain the equivalence class of the interrupted thread and then broadcast it to all the nodes of the application. The node (there can be only one) that has a thread in the equivalence class executing the `wait` operation of our library will then stop waiting and the operation will throw the `InterruptedException`.

To summarize, the J-Orchestra technique for correct monitor-style synchronization of distributed programs in Java addresses the lack of coordination between Java concurrency mechanisms and Java middleware. This technique comprehensively solves the problem and combines the best features of past approaches by enabling distributed synchronization that is both portable and efficient.

4.7 Appletizing: Partitioning for Specialized Domains

Some domains present interesting opportunities for specializing J-Orchestra partitioning. One such domain is a client-server environment in which the client runs as a Java applet that communicates with the server through RMI. We call this specialization *appletizing*, and its primary purpose is adapting legacy Java code for distributed execution. In our context, the term 'legacy' refers to all centralized Java applications, written without distribution in mind, that as part of their evolution need to move parts of their execution functionality to a remote machine. The amount of such legacy code in Java is by no means insignificant with the Java technology being a decade old and four million Java developers worldwide [13].

A large part of what makes Java a language that "allows application developers to write a program once and then be able to run it everywhere on the Internet" [25] are standard distribution technologies over the web. Such Java technologies as applets and servlets have two major advantages: they require little to no explicit distributed programming and they are easily deployable over standard web browsers. Nevertheless, these technologies are inflexible. In the case of applets, a web browser first transfers an applet's code from the server site to the user system and then executes it safely on its JVM, usually in order to draw graphics on the client's screen. In the symmetric case of servlets, code executes entirely on the server, usually in order to access a shared resource such as a database, transmitting only simple inputs and outputs over the network. Therefore, these standard technologies offer a hard-coded answer to the important question of how the distribution should take place, and it is the same for each applet and servlet. Besides these two extremes, one can imagine many other solutions that are customizable for individual programs. A hybrid of the two approaches promises significant flexibility benefits: the programmer can leverage both the resources of the client machine (e.g., graphics, sound, mouse input) and the resources of a server (e.g., shared database, file system, computing power). At the same time, the application will be both safe and efficient: one can benefit from the security guarantees provided by Java applets, while communicating only a small amount of data between the applet and a remote server.

The challenge is to get an approach that runs code both on clients and on a server while avoiding explicit distributed systems development, just like applet and servlet technologies do. Appletizing implements such an approach by semi-automatically transforming a centralized, monolithic Java GUI application into a client-server application, in which the

client runs as a Java applet that communicates with the server through Java RMI. Appletizing builds upon and is a specialization of automatic partitioning with a predefined deployment environment for the resulting client-server applications. Similarly to regular partitioning, appletizing requires no explicit programming or modification to the JVM or its standard runtime classes.

At the same time, the specialized domain makes appletizing more automatic, which required adding several new features to J-Orchestra such as a new static analysis heuristics that automatically assigns classes to the client and the server sites, a profiling heuristic implementation, special bytecode rewrites that adapt certain operations for execution within an applet, and runtime support for the applet/server coordination.

Overall, appletizing offers a unique combination of the following benefits:

- Programming advantages. This includes no-coding distribution and flexibility in writing applications that use complex graphical interfaces and remote resources.
- User deployment advantages. With the client part running as a regular Java applet rather than as a stand-alone distributed application, our approach is accessible to the user via any Java-enabled browser.
- Performance advantages. Appletizing minimizes network traffic through profiling-based object placement and object mobility. This results in transferring less data than when using such remote control technologies as X-Windows.

The advantage of automatic partitioning is that it can put the code near the resource that it controls. In the case of appletizing, partitioning makes it possible to draw graphics locally on the client machine from less data than it takes to transfer the entire graphical representation over the network, while collocating the server resources with the code that controls them. As a special kind of partitioning, appletizing not only offers the same benefits

but also provides a higher degree of automation by enhancing the capacities of several J-Orchestra mechanisms. Next, we describe the specifics of appletizing by detailing the functionality added to static analysis, profiling, bytecode rewriting, and runtime services.

4.7.1 Static Analysis for Appletizing

Consider an arbitrary centralized Java AWT/Swing application that we want to transform into a client-server application through appletizing. First, we classify the application's code (both application classes and the referenced JRE system classes) into four distinct groups, as Figure 4-9 demonstrates schematically.

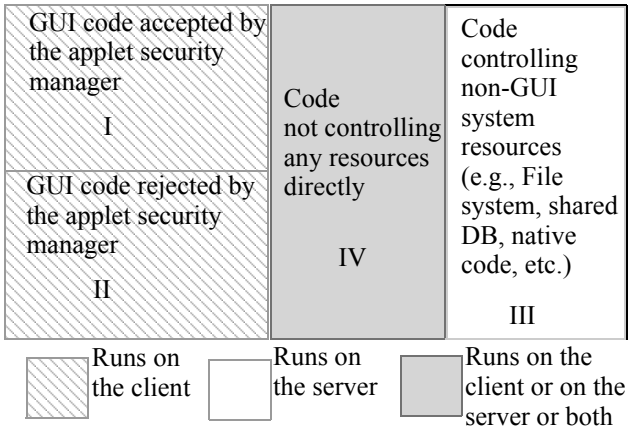


Figure 4-9: The appletizing perspective code view of a centralized Java GUI application.

Group I contains the GUI classes that can safely execute within an applet. Group II contains the GUI classes whose code include instructions that the applet security manager prevents from executing within an applet. For example, an applet cannot perform disk I/O. Group III contains the classes that must execute on the server. The classes in this group control various non-GUI system resources that applets are not allowed to access, such as file I/O operations, shared resources (e.g., a database), and native (JNI) code. Group IV contains

the classes that do not control any system resources directly and as such can be placed on either the client or the server sites, purely for performance reasons. Moreover, objects of classes in this group do not have to remain on the same site during the execution of the program: they can migrate on demand, or according to an application-specific pattern. We implemented the analysis of classes for appletizing on top of the standard J-Orchestra type-based "classification" heuristic (Section 4.4) that groups classes whose instances can be accessed by the same native code.

4.7.2 Profiling for Appletizing

Having completed the aforementioned classification heuristics, J-Orchestra assigns the classes in groups I, II, and III to the client, client, and server sites, respectively. The classification does not offer any help in assigning the classes in group IV, so the user has to do this manually before the rewriting for appletizing can commence. Deciding on the location of a class just by looking at its name can be a prohibitively difficult task, particularly if no source code is available and the user has only a black-box view of the application. To help the user in determining a good placement, J-Orchestra offers an off-line profiler that reports data exchange statistics among different entities (i.e., anchored groups and mobile classes). Integrated with the profiler is a clustering heuristic that, given some initial locations and the profiling results, determines a good placement for all classes. The heuristic is strictly advisory—the user can override it at will. Our heuristic implements a greedy strategy: start with the given initial placement of a few entities and compute the affinity of each unassigned entity to each of the locations. (Affinity to a location is the amount of data exchanged between the entity and all the entities already assigned to the location combined.) Pick the overall maximum of such affinity, assign the entity that has it to the corresponding location

and repeat until all entities are assigned. In principle, appletizing offers more opportunities than general application partitioning for automation in clustering: optimal clustering for a client-server partitioning can be done in polynomial time, while for 3 or more partitions the problem is NP-hard [24]. In practice we have not yet had the need to replace our heuristic for better placement.

In terms of implementation, the J-Orchestra profiler has evolved through several incarnations. The first profiler worked by instrumenting the Java VM through the JVMPI and JVMDI (Java Virtual Machine Profiling/Debugging Interface) binary interfaces. We found the overheads of this approach to be very high, even for recent VMs that enable compiled execution under debug mode. The reason is the "impedance mismatch" between the profiling code (which is written in C++ and compiled into a dynamic library that instruments the VM) and the Java object layout. Either the C++ code needs to use JNI to access object fields, or the C++ code needs to call a Java library that will use reflection to access fields. We have found both approaches to be much slower (15x) than using bytecode engineering to inject our own profiling code in the application. The profiler rewrite is isomorphic to the J-Orchestra rewrite, except that no distribution is supported—proxies keep track of the amount of data passed instead.

An important issue with profiling concerns the use of off-line vs. on-line profiling. Several systems with goals similar to ours (e.g., Coign [33] and AIDE [56]) use on-line profiling in order to dynamically discover properties of the application and possibly alter partitioning decisions on-the-fly. So far, we have not explored an on-line approach in J-Orchestra, because of its overheads for regular application execution. Since J-Orchestra has no control over the JVM, these overheads can be expected to be higher than in other sys-

tems that explicitly control the runtime environment. Without low-level control, it is hard to keep such overhead to a minimum. Sampling techniques can alleviate the overhead (at the expense of some accuracy) but not eliminate it: some sampling logic has to be executed in each method call, for instance. We hope to explore the on-line profiling direction in the future.

4.7.3 Rewriting Bytecode for Appletizing

Once all the classes are assigned to their destination sites, J-Orchestra rewrites the application for appletizing, which augments the regular J-Orchestra rewrite with an additional step that modifies unsafe instructions in GUI classes for executing within an applet. The applet security manager imposes many restrictions on what resources applets can access, and many of these restrictions affect GUI code. J-Orchestra inspects the bytecode of an application for problematic operations and "sanitizes" them for safe execution within an applet. Depending on the nature of an unsafe operation, J-Orchestra uses two different replacement approaches. The first approach replaces an unsafe operation with a safe, semantically similar (if not identical) version of it. For example, an unsafe operation that reads a graphical image from disk gets rewritten with a safe operation that reads the same image from the applet's jar file. The following code fragment demonstrates the rewrite. (We use source code only for ease of exposition—all modifications take place at the bytecode level):

```
//Creates an ImageIcon from the specified file
//will cause a security exception
//when a file on disk is accessed
javax.swing.ImageIcon icon =
           new javax.swing.ImageIcon ("SomeIconFile.gif");
```

```
//Sanitize by replacing with the following safe code
javax.swing.ImageIcon icon =
    new jorchestra.runtime.ImageIcon ("SomeIconFile.gif");

//The implementation of jorchestra.runtime.ImageIcon
//constructor
//(part of J-Orchestra runtime functionality)
public ImageIcon (String fileName) {
 //obtain and pass a URL to the super constructor
 //of javax.swing.ImageIcon
 super (getURL (fileName)); //will safely read the image
                            //from the applets's jar file
}
```

The second approach, replaces an unsafe operation with a semantically different operation. For example, since applets are not allowed to call `System.exit`, this method call gets replaced with a call to the J-Orchestra runtime service that informs the user that they can exit the applet by directing the web browser to another page. Sometimes, replacing an unsafe instruction requires a creative solution. For example, the applet security manager prevents the `setDefaultCloseOperation` method in class `javax.swing.JFrame` from accepting the value `EXIT_ON_CLOSE`. Since we cannot change the code inside `JFrame`, which is a system class, we modify the caller bytecode to pop the potentially unsafe parameter value off the stack and push the safe value `DO_NOTHING_ON_CLOSE` before calling `setDefaultCloseOperation`. The following code snippet demonstrates the specifics of this bytecode replacement.

```
//the following two instructions are inserted
//before every invocation
//of setDefaultCloseOperation method of javax.swing.JFrame

pop //pop whatever value on top of the stack
push 0 //param 0 means DO_NOTHING_ON_CLOSE
invokevirtual javax.swing.JFrame.setDefaultCloseOperation
```

```
//If there are any jumps whose target is
//the invokevirtual instruction,
//they are redirected to the pop instruction instead.
```

Once unsafe instructions in GUI classes have been replaced, J-Orchestra proceeds with its standard rewrite that ends up packaging all the rewritten classes in client and server jar files ready for deployment.

The GUI-intensive nature of appletizing also allows us to perform special-purpose code transformations to optimize remote communication. For instance, knowing the design principles of the Swing/AWT libraries allows us to pass Swing event objects using by-copy semantics. This is done by making event objects implement `java.io.Serializable` and adding a default no arguments constructor if it is not already present. Passing event objects by-copy is typically safe because event listener code commonly uses event objects as read-only objects, since the programming model makes it very difficult to determine in what order event listeners receive events.

Currently the rewrite does not fully maintain the Swing design invariant of having all event-dispatching and painting code execute in a single event-dispatching thread. This can make a graphical application execute incorrectly when partitioned for distributed execution. The problem is that splitting a single-threaded application into a client and server parts creates implicit multithreading. Thus, the server could call client Swing code remotely through RMI on a thread different from the event-dispatching one. Figure 4-10 demonstrates pictorially how this situation could occur. This is a so-called zig-zag calling pattern, in which a GUI calls `someServerMethod` on the server. Then `someServer-Method`, in response, calls back `someGUIOp` method on the client, which is invoked on a new thread, different from the one designated for event dispatching. As we have seen in

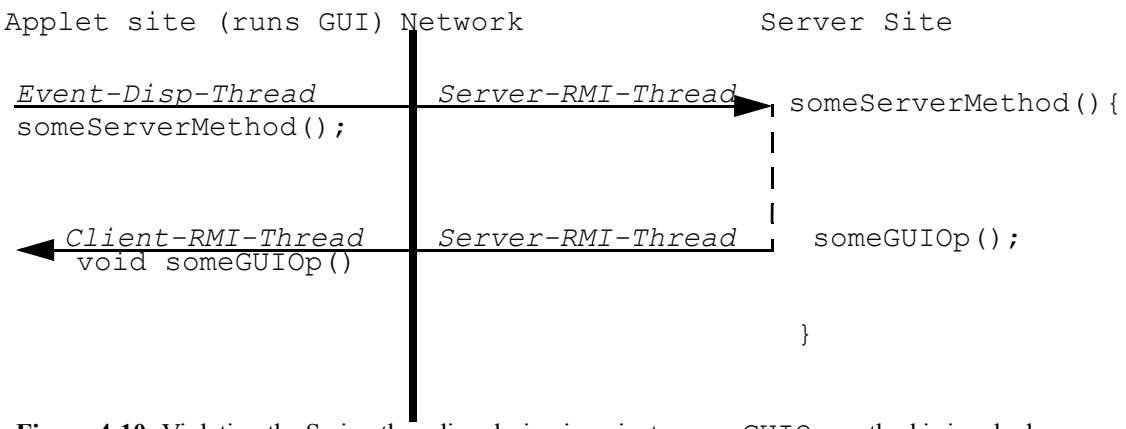


Figure 4-10: Violating the Swing threading design invariant: `someGUIOp` method is invoked on a thread different than *Event-Disp-Thread,* if no special care is taken.

Section 4.6, this happens because Java RMI does not preserve thread identity for executions spanning multiple machines. The J-Orchestra approach to maintaining concurrency and synchronization over RMI simply treats different threads (e.g., *Event-Disp-Thread*, *Server-RMI-Thread*, and *Client-RMI-Thread* in this case) as belonging to the same equivalence class and as such cannot ensure that all GUI code is executed in a single designated event-dispatching thread. Therefore, appletizing must enable special-purpose handling of Swing code.

A fully general solution would ensure that the event-dispatching thread never blocks while making a remote call. This can be accomplished by spawning a new thread for each remote call made in the event-dispatching thread. Then the event-dispatching thread would just wait for two kinds of events: either for the spawned thread to complete or for newly-arrived GUI events (e.g., screen updates) that need to take place. If the server makes a remote call to a GUI method, the method will just add a new GUI event to a queue and will notify the event-dispatching thread to execute it, while the server is blocked—just as in the

local execution case. We plan to implement this faithful emulation of the local execution semantics in the future. Currently, however, we only offer a simpler, approximate solution that handles a special case of the problem. We also report to the user all instances of GUI methods potentially called by the server part of the application, since we do not transparently guarantee correct execution in all cases.

Specifically, our current solution works only for GUI methods that return `void` and do not change the state of the application in any way other than by producing graphical output. In most cases, when the backend calls the front end GUI, it does so through the so called callback methods that just perform some drawing actions and do not return any values. The current implementation uses the existing Swing facility (`SwingUtilities.invokeLater` method) to enable any thread to request that the event-dispatching thread runs certain code. The following is the code in the translator of some GUI class for method `someGUIOp` from Figure 4-7, executing at the client site:

```
//make all parameters final, to be able to pass them
//to the anonymous inner class
public void someGUIOp (final int param) throws
RemoteException {

 SwingUtilities.invokeLater(new Runnable () {
   public void run () {
     _localObj.someGUIOp(param);
   }
 });
}
```

In other words, whenever a `void`-returning method performs operations using the Swing library (perhaps transitively, through other method calls), we make sure that remote calls of the method result in its delayed execution inside the event dispatching thread. The actual GUI code will be executed only when the remote call returns. We have found this

incomplete solution sufficient for successfully appletizing the applications described in our case studies in Section 5.7.1.

Note that both the current partial solution and a future full emulation of local Swing semantics fit well in our appletizing techniques. Recall the structure of the J-Orchestra indirection approach: classes that are co-anchored on the same site, such as the applet's GUI classes, end up calling each other directly. The server classes, on the other hand, can call client classes (and vice versa) only through a proxy/translator chain. Thus, all events that we need to trap (namely, remote calls inside the event-dispatching thread and remote invocations of a GUI method) are handled through a translator—hence, only the code inside the generated translator classes needs to change. This is simple, as it requires no modification of the existing binary code of the application, and imposes no overhead on the local execution of methods.

4.7.4 Runtime Support for Appletizing

Appletizing works with standard Java-enabled browsers that download the applet code from a remote server. To simplify deployment, the downloaded code is packaged into two separate jar files, one containing the application classes that run on the client and the other J-Orchestra runtime classes. In other words, the client of an appletized application does not need to have pre-installed any J-Orchestra runtime classes, as a Java-enabled browser downloads them along with the applet classes. Once the download completes, the J-Orchestra runtime client establishes an RMI connection with the server and then invokes the main method of the application through reflection. The name of the application class that contains the main method along with the URL of the server's RMI service are supplied as applet parameters in an automatically generated HTML file. Figure 4-11 shows such an

```
<html>
 <head><title>Jarminator run as a J-Orchestra Applet</title>
 </head>
 <body>
 <APPLET WIDTH=1 HEIGHT=1
   CODE="jorchestra/runtime/applet/Applet.class"
   ARCHIVE="jarminator.jar, jorchestra.jar" >
   <PARAM NAME="CLASSNAME"
      VALUE="remotecapable.net.weird173.jarminator.Jarminator">
  <PARAM NAME="CLIENT_NODE_NAME" VALUE="jarminator_client">
  <PARAM NAME="SERVER_NODE_NAME" VALUE="jarminator_server">
 </APPLET>
 </body>
</html>
```

Figure 4-11: An automatically generated HTML file for deploying the appletized Jarminator application.

HTML file for one of our case studies, which is discussed in-detail in Section 5.7.1. This arrangement allows hosting multiple J-Orchestra applets on the same server that can share the same set of runtime classes. In addition, multiple clients can simultaneously run the same applet, but they will also spawn distinct server components. Our approach cannot make an application execute concurrently when it was not designed to do so. In addition to communication, the J-Orchestra applet runtime provides various convenience services such as access to the properties of the server JVM, a capacity for terminating the server process, and a facility for browsing the server's file system efficiently.

Chapter V presents several case studies of successfully appletizing realistic, third-party applications, confirming the benefits of the approach. Specifically, our measurements show that the appletized applications perform favorably both in terms of network traffic and overall responsiveness compared to using a remote X display for controlling and monitoring the applications. Taking these results into account, it is safe to say that appletizing,

having the benefits of automation, flexibility, ease of deployment, and good performance, can be a useful tool for software evolution.

4.8 Run-Time Performance

This section examines issues of run-time performance of programs partitioned with J-Orchestra. First of all, it is important to state that the performance characteristics of a partitioned application depend primarily on the ability to derive a good placement for anchored groups and to determine performance-improving object mobility scenarios. In that respect, it is the user who has to estimate the potential data exchange patterns between network sites, possibly assisted by the J-Orchestra profiling tool. Thus this section is not concerned with estimating overheads caused by bad partitioning decisions. Rather it looks at the indirection overheads specific to the J-Orchestra rewrite. Although anchoring by choice can practically eliminate the indirection overheads of the J-Orchestra rewrite, it is worth examining how high these overheads can be in the worst case. Section 4.8.1 presents measurements of these overheads and details the local-only optimization employed in J-Orchestra.

4.8.1 Indirection Overheads and Optimization

4.8.1.1 Indirection Overheads

The most significant overheads of the J-Orchestra rewrite are one level of indirection for each method call to a different application object, two levels of indirection for each method call to an anchored system object, and one extra method call for every direct access to another object's fields. The J-Orchestra rewrite keeps overheads as low as possible. For instance, for an application object created and used only locally, the overhead is only one interface call for every virtual call, because proxy objects refer directly to the target object

and not through RMI. Interface calls are not expensive in modern JVMs (only about as much as virtual calls [2]) but the overall slowdown can be significant.

The overall impact of the indirection overhead on an application depends on how much work the application's methods perform per method call. A simple experiment puts the costs in perspective. Figure 4-4 shows the overhead of adding an extra interface indirection per virtual method call for a simple benchmark program. The overall overhead rises from 17% (when a method performs 10 multiplications, 10 increment, and 10 test operations) to 35% (when the method only performs 2 of these operations).

Table 4-4. J-Orchestra worst-case indirection overhead as a function of average work per method call (a billion calls total)

Work (multiply, increment, test)	Original Time	Rewritten Time	Overhead
2	35.17s	47.52s	35%
4	42.06s	51.30s	22%
10	62.5s	73.32s	17%

Penalizing programs that have small methods is against good object-oriented design, however. Furthermore, the above numbers do not include the extra cost of accessing anchored objects and fields of other objects indirectly (although these costs are secondary). To get an idea of the total overhead for an actual application, we measured the slowdown of the J-Orchestra rewrite using J-Orchestra itself as input. That is, we used J-Orchestra to translate the main loop of the J-Orchestra rewriter, consisting of 41 class files totalling 192KB. Thus, the rewritten version of the J-Orchestra rewriter (as well as all system classes it accesses) became remote-capable but still consisted of a single partition. In local execution, the rewritten version was about 37% slower (see Figure 4-5 later). Although a 37% slowdown of local processing can be acceptable for some applications, for many others it

is too high. Recall, however, that this would be the overhead of the J-Orchestra rewrite for a partitioning where all application objects were mobile. Anchoring by choice all but a few mobile classes completely eliminates this overhead.

4.8.1.2 Local-Only Optimization

Recall that remote objects extend the RMI class `UnicastRemoteObject` to enable remote execution. The constructor of `UnicastRemoteObject` exports the remote object to the RMI run-time. This is an intensive process that significantly slows down the overall object creation. J-Orchestra tries to avoid this slowdown by employing lazy remote object creation for all the objects that might never be invoked remotely. If a proxy constructor determines that the object it wraps is to be created on the local machine, then the creation process does not go through the object factory. Instead, a *local-only* version of the remote object is created directly. A local-only object is isomorphic to a remote one but with a different name and without inheriting from `UnicastRemoteObject`. A proxy continues to point to such a local-only object until the application attempts to use the proxy in a remote method call. In that case, the proxy converts its local-only object to a remote one using a special conversion constructor. This constructor reassigns every member field from the local-only object to the remote one. All static fields are kept in the remote version of the object to avoid data inconsistencies.

Although this optimization may at first seem RMI-specific, in fact it is not. Every middleware mechanism (even such recent standards as .NET Remoting) suffers significant overhead for registering remotely accessible objects. Lazy remote object creation ensures that the overhead is not suffered until it is absolutely necessary. In the case of RMI, our experiments show that the creation of a remotely accessible object is over 200 times more

expensive than a single constructor invocation. In contrast, the extra cost of converting a local-only object into a remotely accessible one is about the same as a few variable assignments in Java. Therefore, it makes sense to optimistically assume that objects are created only for local use, until they are actually passed to a remote site. Considering that a well-partitioned application will only move few objects over the network, the optimization is likely to be valuable.

The impact of speeding up object creation is significant in terms of total application execution time. We measured the effects using the J-Orchestra code itself as a benchmark. The result is shown below (Figure 4-5). The measurements are on the full J-Orchestra rewrite: all objects are made remote-capable, although they are executed on a single machine. 767 objects were constructed during this execution. The overhead for the version of J-Orchestra that eagerly constructs all objects to be remote-capable is 58%, while the same overhead when the objects are created for local use is less than 38% (an overall speedup of 1.15, or 15%).

Table 4-5. Effect of lazy remote object creation (local-only objects) and J-Orchestra indirection

Original time	Indirect lazy	Overhead	Indirect non-lazy	Overhead
6.63s	9.11s	37.4%	10.48s	58.1%

4.9 Java Language Features And Limitations

J-Orchestra needs to handle many Java language features specially in order to enable partitioning of unsuspecting applications. Features with special handling include inheritance, static methods and fields, object creation, arrays, object identity, synchronization, reflection, method access modifiers, garbage collection, and inner classes. We do not

describe the low-level specifics of dealing with every language feature here, as they are mostly straightforward—the interested reader should consult this publication [87] for more details. Nevertheless, it is interesting to survey some of the limitations of the system, both in its safety guarantees and in offering a complete emulation of a single Java VM over a distributed environment.

4.9.1 Unsafety

As mentioned in Section 4.4, there will always be unsafeties in the J-Orchestra classification, but these are inherent in the domain of automatic partitioning and not specific to J-Orchestra. No partitioning algorithm will ever be safe without assumptions about (or analysis of) the platform-specific binary code in the system classes. System code can always behave badly, keeping aliases to any object that gets created and accessing its fields directly, so that no proxy can be used instead. Additionally, several objects are only created and used implicitly by native code, without their presence ever becoming explicit at the level of the interface between system and application code. For example, every site is implicitly associated with at least one thread object. If the application semantics is sensitive to all threads being created on the same machine, then the execution of the partitioned application will not be identical to the original one. Similarly, every JVM offers predefined objects like `System.in`, `System.out`, `System.err`, `System.properties` and `System.exit`. The behavior of an application using these stateful implicit objects will not be the same on a single JVM and on multiple ones. Indeed, it is not even clear that there is a single correct behavior for the partitioned application—different policies may be appropriate for different scenarios. For example, when one of the partitions writes something to the standard output stream, should the results be visible only on the network site of

the partition, all the network sites, or one specially designated network site that handles I/O? If one of the partitions makes a call to `System.exit`, should only the JVM that runs that partition exit or the request should be applied to all the remaining network sites? J-Orchestra allows defining these policies on a per-application basis.

4.9.2 Conservative classification

The J-Orchestra classification is quite conservative. For instance, it is perfectly reasonable to want to partition an application so that two different sites manipulate instances of a certain anchored unmodifiable class. For example, two different machines may need to use graphical windows, but without the windows manipulated by code on one machine ever leaking to code on the other. J-Orchestra cannot tell this automatically since it has to assume the worst about all references that potentially leak to native code. Thus, partitionings that require objects of the same anchored unmodifiable class to be created on two different sites are not safe according to the J-Orchestra classification. This is a problem that is commonly encountered in practice. In those cases, the user needs to manually override the J-Orchestra classification and assert that the classes can safely exist on two sites. Everything else proceeds as usual: the translation wrapping/unwrapping technique is still necessary, as it enables indirect access to anchored unmodifiable objects (e.g., so that code on site *A* can draw on a window of site *B*, as long as it never passes the remote reference to local unmodifiable code).

4.9.3 Reflection and dynamic loading

Reflection can render the J-Orchestra translation incorrect. For instance, an application class may get an `Object` reference, query it to determine its actual type, and fail if the

type is a proxy. Nevertheless, the common case of reflection that is used only to invoke methods of an object is compatible with the J-Orchestra rewrite—the corresponding method will be invoked on the proxy object. Similar observations hold regarding dynamic class loading. J-Orchestra is meant for use in cases where the entire application is available and gets analyzed, so that the J-Orchestra classification and translation are guaranteed correct. Currently, dynamically loading code that was not rewritten by J-Orchestra may fail because the code may try to access remote data directly. Nevertheless, one can imagine a loader installed by J-Orchestra that takes care of rewriting any dynamically loaded classes before they are used. Essentially, this would implement the entire J-Orchestra translation at load time. Unfortunately, classification cannot be performed at load time. The J-Orchestra classification is a whole-program analysis and cannot be done incrementally: unmodifiable classes may be loaded and anchored on some nodes before loading another class makes apparent that the previous anchorings are inconsistent.

4.9.4 Inherited limitations

J-Orchestra inherits some limitations from its underlying middleware—Java RMI. These limitations are better addressed uniformly at the middleware level than by J-Orchestra. One limitation has to do with efficiency. Although RMI efficiency has improved in JDK 1.4, RMI still remains a heavyweight protocol. Another limitation concerns distributed garbage collection. J-Orchestra relies on the RMI distributed reference counting mechanism for garbage collection. This means that cyclic garbage, where the cycle traverses the network, will never be collected.

Additionally, J-Orchestra does not explicitly address security and administrative domain issues—indeed the J-Orchestra rewrite even weakens the protections of some

methods, e.g., to make them accessible through an interface. We assume that the user has taken care of security concerns using an orthogonal approach to establish a trusted domain (e.g., a VPN).

4.10 Conclusions

Accessing remote resources has now become one of the primary motivations for distribution. In this chapter we have shown how J-Orchestra allows the partitioning of programs onto multiple machines without programming. Although J-Orchestra allows programmatic control of crucial distribution aspects (e.g., handling errors related to distribution) it neither attempts to change nor facilitates changing the structure of the original application. Thus, J-Orchestra is applicable in cases in which the original application has loosely coupled parts, as is commonly the case of controlling multiple resources.

Although J-Orchestra is certainly not a "naive end-user" tool, it is also not a "distributed systems guru" tool. Its ideal user is the system administrator or third-party programmer who wants to change the code and data locations of an existing application with only a superficial understanding of the inner workings of the application.

We believe that J-Orchestra is a versatile tool that offers practical value and interesting design ideas. J-Orchestra is interesting on the technical front as the first representative of partitioning tools with what we consider important characteristics:

- use of a high-level language runtime, such as the Java VM or the Microsoft CLR; performing modifications directly at the binary level.
- no changes to the runtime required for partitioning.

- provisions for correct execution even in the presence of code unaware of the distribution (e.g., Java system code).

While this chapter has concentrated on the motivation, design, and implementation issues of J-Orchestra, Chapter V looks at the applicability of the automatic partitioning approach and presents several case studies of successfully partitioning various applications.

CHAPTER V

APPLICABILITY AND CASE STUDIES

This chapter argues that this dissertation explores algorithms, techniques, and tools for separating distribution concern that can be a valuable addition to the working programmer's toolset. The content of this chapter, as its title suggests, is divided into two parts: discussing general applicability issues followed by supporting our claims through a series of case studies. The discussion starts by showing how NRMI, GOTECH, and J-Orchestra compare with each other and how they follow the translucent approach to separating distribution concerns. Then we take a closer look at each software tool from the applicability perspective. This includes identifying for each tool the distribution concerns that it separates, the common architectural characteristics of the centralized applications that it can effectively and efficiently transform for distributed execution, and the programming scenarios under which programmers are most likely to find it useful. The second part of the chapter presents a series of case studies that apply each of the three tools to various programming scenarios, thus supporting our applicability claims empirically.

5.1 Applicability of the Translucent Approach

This dissertation explores software tools that separate distribution concerns by following the approach we call "translucent." This approach, while aiming at distribution transparency, nevertheless, does not attempt to mask all the differences between the cen-

tralized and distributed execution models. NRMI—a middleware mechanism, GOTECH—a code generator, and J-Orchestra an automatic partitioning system demonstrate the value of the translucent approach in their own categories. On the scale of automation, NRMI is the least automatic tool, being just a middleware mechanism, GOTECH introduces distribution into centralized programs, given the programmer's annotations, and J-Orchestra automates the entire distribution process, requiring only high-level GUI-based input from the programmer. Next we describe in greater detail how each of these tools follows the translucent approach.

NRMI makes remote calls look like local calls as far as the parameters passing semantics is concerned for stateless servers and single-threaded clients. It is the programmer's responsibility to ensure that these preconditions hold true. At the same time, NRMI does not hide the possibility of partial failures, and similarly to regular RMI, every remote call can throw various remote exceptions, and the programmer is responsible for supplying custom code for catching and handling them. Thus, with NRMI and call-by-copy-restore, remote calls have a closer semantics to the one of local calls without trying to hide the possibility of partial failure.

GOTECH uses NRMI as its building block, but has more preconditions for successful application. Prior to specifying which local calls GOTECH should transform into remote calls, the programmer has to be aware that the structure of the original application is amenable to such a transformation. This includes not only the preconditions for the successful application of NRMI (i.e., a stateless server and a single-thread client) but also that the centralized program follows the strict client-server communication model. The programmer is also responsible for providing custom code for handling the remote exceptions

that could be raised during every remote method invocation. Thus, GOTECH relieves the programmer from having to write tedious and error-prone distribution code by hand, but the code that it generates does not attempt to hide the fact that the distribution has taken place.

Finally, J-Orchestra is the most automatic of the tools and also separates the largest number of distribution concerns. Nevertheless, J-Orchestra is not a distributed shared memory system and aims at functional distribution, putting the code near the resources it manages. Despite having some preconditions for successful application that are discussed in Section 5.4, J-Orchestra works correctly with a very broad subset of the Java language and shares none of the preconditions of NRMI and GOTECH. Specifically, a distributed program, created through J-Orchestra partitioning, can have stateful servers and multi-threaded clients. As far as the possibility of failures is concerned, J-Orchestra uses regular Java RMI, and a sophisticated user can provide custom error handling in response to raised remote exceptions. Thus, J-Orchestra relieves the programmers from having to figure out all the complex data sharing scenarios done through references, allowing them to concentrate on the challenging issues in distributed computing such as handling partial failures.

Next we take a closer look at the applicability issues of NRMI, GOTECH, and J-Orchestra in turn.

5.2 Applicability of NRMI: Usability Call-by-Copy-Restore vs. Call-by-Copy

Compared to call-by-copy, call-by-copy-restore semantics offers better usability, for it simulates the local execution semantics very closely, as discussed in Section 2.4.1.

Clearly, call-by-copy-restore semantics can be achieved by using call-by-copy and adding application-specific code to register and re-perform any updates necessary. Nevertheless, taking this approach has several disadvantages:

- The programmer has to be aware of all aliases in order be able to update the values changed during the remote call, even if the changes are to data that became unreachable from the original parameters.

- The programmer needs to write extra code to perform the update. This code can be long for complex updates (e.g., up to 100 lines per remote call for the microbenchmarks we discuss in Section 5.5.3).

- The programmer cannot perform the updates without full knowledge of what the server code changed. That is, the changes to the data have to be part of the protocol between the server programmer and the client programmer. This complicates the remote interfaces and specifications.

As we discussed in Section 2.2 on page 15, a call-by-copy-restore semantics is most valuable in the presence of aliased data. Aliasing occurs as a result of several common implementation techniques in mainstream programming languages. All of these techniques produce code that is more convenient to write using call-by-copy-restore middleware than using call-by-copy middleware. Specific examples include:

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

- A variant of this pattern occurs when GUI elements (e.g., menus, toolbars) hold aliases to program data that can be modified. The reason for multiple aliasing is that the same

data may be visible in multiple toolbars, menus, and so forth or that the data may need to be modified programmatically with the changes reflected in the menu or toolbar. For example, we distribute with NRMI a modified version of one of the Swing API example applications. We changed the application to be able to display its text strings in multiple languages. The change of language is performed by calling a remote translation server when the user chooses a different language from a drop-down box. (That is, the remote call is made in the event dispatching thread, conforming to Swing thread programming conventions [81].) The remote server accepts a vector of words (strings) used throughout the graphical interface of the application and translates them between English, German, and French. The updated list is restored on the client site transparently, and the GUI is updated to show the translated words in its menus, labels, and so on. The distributed version code (using the RMI drop-in replacement implementation) has only two tiny changes compared to local code: a single class needs to implement the NRMI marker interface `java.rmi.Restorable`, and a method has to be looked up using a remote lookup mechanism before getting called. In contrast, the version of the application that uses regular Java RMI has to use a more complex remote interface for getting back the changed data and the programmer has to write code in order to perform the update.

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

5.3 Applicability of GOTECH: What are the Distribution Concerns and Can They Be Separated?

For insight into identifying "distribution concerns," we refer to the "differences" between local and distributed models of computation listed in Waldo et al.'s well-known "A Note on Distributed Computing" [96]. The distribution concerns that GOTECH aims to separate fall into three main groups: semantics, performance, and conventions.

5.3.1 Semantics

Consider a centralized application written in a modular fashion with separate objects handling distinct parts of the functionality of the application. It might seem that moving a part of the functionality to a remote machine is just a matter of making some object remotely accessible by the rest of the application. Nevertheless, objects can be sharing data through memory references (which are valid only in a single address space). Of course, one could emulate a single address space over a network of nodes by making all references be over the network. However, such an emulation would be prohibitively slow.

As a result, the semantics of remote method calls are different from the semantics of local calls under standard middleware. That is, the same code will behave differently if executed in the same process and if executed as a remote call (using CORBA, RMI, DCOM, and so forth) on a different machine. Therefore, the lack of a shared address space is the single most important conceptual difference introduced by distribution. This problem cannot be solved in a fully general way. For instance, an application may have a structure such that all its parts are tightly coupled, access each other's data (or OS-level resources, like I/O) directly and depend on reading the latest values of these data. In this case, efficient

distribution is impossible without a change in the application structure. Therefore, the assumption of the GOTECH approach is that the application is amenable to added distribution without a fundamental change in the application structure. In this case, the memory semantics issue can be alleviated if the programmer has control over the calling semantics and if local semantics can be emulated under certain assumptions so that the programmer does not need to write a lot of tedious code. Distribution also requires changes to the client of a remote object to become aware of the possibility of partial failures. Again, this problem cannot be solved in a fully general way, but Java language designers used the exception mechanism to ensure partial failure awareness: the client of a remote call needs to handle various exceptions that might arise in response to various partial failure conditions.

5.3.2 Performance

With processor speed continuing to increase at a much higher rate than network performance, remote calls have become more costly than ever compared to local calls. When some local calls suddenly become remote, the resulting distributed application may become unusable due to a slowdown by orders of magnitude. When applying distribution as a separate step, one has to be aware of such latencies when deciding whether an object can be moved to a remote site. An object can be moved to a remote site only if it is not tightly coupled with the rest of the application. For this reason, the programmer should have complete control over the location of objects.

5.3.3 Conventions

Using a middleware mechanism such as RMI, CORBA, DCOM, and so forth to enable distribution has become a common business practice. Since GOTECH aims to

remove the low-level technical barriers to aspectizing distribution, our main challenge is to change application code so that it interfaces with distribution middleware, which entails manipulating code according to established conventions. In object-oriented distributed systems, types are often used to mark an object that can interact with the middleware runtime services. For example, to interact with such a runtime service an object might have to implement certain interfaces by providing methods that are called by the middleware at runtime. Another example would be to declare the remote methods of an object as throwing exceptions for potential network errors. The client code requires some modifications as well. A call to a remote object constructor might have to be replaced by a sequence of calls to a registry service. Making such changes can be quite tedious. Tool vendors have made some inroads in alleviating the task of converting plain objects so that they conform to a given framework convention. One such example is Microsoft's Class Wizard for Visual C++, which creates an MFC class from a given COM object. However, none of these industrial tools help the programmer make changes to the *clients* of the modified object.

The authors of the “Note” suggest that, “providing developers with tools that help manage the complexity of handling the problems of distributed application development as opposed to the generic application development is an area that has been poorly addressed.” Hopefully, the ideas introduced by GOTECH can help in providing developers with such tools.

5.4 Applicability of J-Orchestra: Conditions for Successful Automatic Partitioning

"It's not how well the bear dances;
it's that it can dance at all"

J-Orchestra can handle a large subset of Java and, thus, can correctly partition a large class of realistic unsuspecting applications. Nevertheless, among these, J-Orchestra will be useful only in a few well-defined cases. Automatic partitioning is not a substitute for general distributed systems development. The striking element of the approach is not that it is widely applicable but that it is at all applicable, given how automated it is.

We introduce the term *embarrassingly loosely coupled* to describe the kinds of applications to which J-Orchestra is applicable. An embarrassingly loosely coupled application satisfies two criteria:

- it has components that exchange little data with the rest of the application, and
- these components are statically identifiable by looking at the structure of the application code at the class or the module level.

That is, by looking at static relations among application classes, the user of J-Orchestra (aided by our analysis tools) should be able to identify distinct components comprising multiple classes. Then, during run-time, the data coupling among distinct components should be very small. In other words, an application should have very clear communication and locality patterns. Since the application logic will remain the same, a large number of remote accesses will be detrimental to performance. This requirement stems from the intrinsic difference in latency between local and remote method calls. As an

illustration, consider a sequence of method calls constituting an execution scenario. In a centralized, monolithic application, an execution scenario comprises local method calls, executing in a shared address space. When partitioning takes place, in the same execution scenario some of the method calls become remote, invoked through an RPC mechanism such as Java RMI. As the latency of a remote call is several orders of magnitude larger than that of a local call, a partitioned version of a sequential centralized application is expected to take longer to execute.[1]

The slowdown factor can be defined as the difference in total execution time (disregarding such additional factors as waiting for user input or interacting with the OS) between the original centralized application and its partitioned version. Of course, applications differ in terms of how much slowing down they can afford as a result of partitioning before the process would render them unusable. Nevertheless, a performance optimization, aimed at reducing the total number of remote calls, would be beneficial for any application. What determines the total number of remote calls made by a particular partitioned application is not just the initial static class placement but also various object mobility scenarios. Many modern networks, in which latency is more of an issue than bandwidth, present optimization opportunities via the means of object mobility (i.e., moving method arguments to or the return value from the site of a remote call) that could reduce the total number of remote calls. Approaches to estimating the expected slowdown factor of an application include online or offline profiling that could take into consideration both the initial static placement of objects and object mobility scenarios. Once presented with the profiling results, the pro-

1. An additional slowdown results from the fact that a partitioned version of a centralized application always ends up making more (local) method calls in a given execution scenario due to such mechanisms as proxy indirection and object factory lookup introduced by the partitioning.

grammer can decide whether the expected slowdown factor is acceptable for a given application.

Once partitioned, an embarrassingly loosely coupled application must not share objects among partitions that are used by unmodifiable code (e.g., OS or JVM code) and should have synchronous communication patterns. If either of these two properties do not hold true or if good performance or reliability requires asynchronous communication, the application structure needs to change.

Hence, embarrassingly loosely coupled applications can be partitioned automatically without significant loss in performance due to network communication. However, in order to get any benefit, the application needs to have a reason to be distributed. The foremost reason for distributing an application with J-Orchestra is to take advantage of remote hardware or software resources (e.g., a processor, a database, a graphical screen, or a sound card). Several special-purpose technologies do this already: distributed file systems allow storage on remote disks; remote desktop applications (e.g., VNC [69], X [70]) allow transferring graphical data from a remote machine; network printer protocols let users print remotely. Nevertheless, the advantage of automatic partitioning is that it can put the code near the resource that it controls. For instance, if a graphical representation can be computed from less data than it takes to transfer the entire graphical representation over the network, then J-Orchestra has an advantage. Some mainstream technologies put code near a resource such as Java applets, which move graphics-producing code from a server to a client with the screen on which the graphics will be displayed. However, this solution is inflexible, as it requires the entire program to move across the network. In contrast, applet-

izing (Section 4.7), a specialization of automatic partitioning, can split an application so that any part of it can become a "virtual applet" and can run on a client machine.

Of course, one reason to partition an application is to take advantage of parallelism. Distinct machines will have distinct CPUs. If the original centralized application is multi-threaded, we can use multiple CPUs to run threads in parallel. Although distribution-for-parallelism is a potential application of J-Orchestra, we have not examined this space so far. The reason is that parallel applications either are written to run in distributed memory environments in the first place, or have tightly coupled concurrent computations.

To summarize, we can characterize the domain of J-Orchestra as *partitioning embarrassingly loosely coupled applications for resource-driven distribution*.

5.5 NRMI Case Studies

Before presenting the results of NRMI performance experiments, we describe the performance optimizations that we applied to the RMI replacement implementation of NRMI. These optimizations demonstrate that *copy-restore middleware can be optimized for real-world use*, which is the main point of our experiments. Although NRMI emphasizes usability, its implementation can be quite efficient: the RMI replacement implementation is optimized and suffers only small overheads. The optimized NRMI for JDK 1.4 is about 20% slower than regular RMI for JDK 1.4. To put this number in perspective, this also means that NRMI for JDK 1.4 is about 20-30% faster than regular RMI for the previous version, JDK 1.3.

5.5.1 NRMI Low-Level Optimizations

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

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

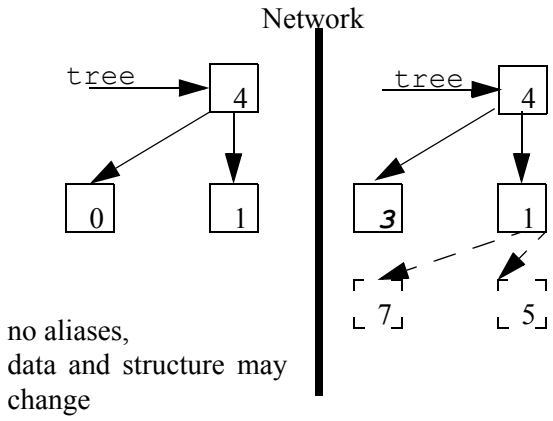
The *optimized* version of NRMI only works with JDK 1.4 and 1.5 and takes advantage of special features exported by the JVM in order to achieve better performance. The performance of regular Java RMI improved significantly between versions 1.3 and 1.4 of the JDK (as we show in our measurements). The main reason was the flattening of the layers of abstraction in the implementation. Specifically, object serialization was optimized through non-portable direct access to objects in memory through an "Unsafe" class exported by the Java Virtual Machine. The optimized version of NRMI also uses this facility to quickly inspect and change objects.

5.5.2 Description of Experiments

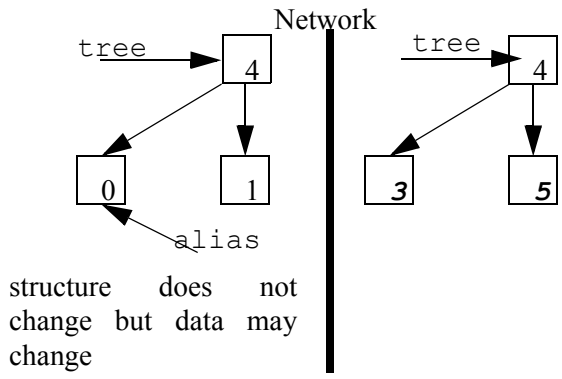
In order to see how our implementation of call-by-copy-restore measures up against the standard implementation of RMI, we created three micro-benchmarks. Each benchmark consists of a single randomly-generated binary tree parameter passed to a remote method. The remote method performs random changes to its input tree. *The invariant maintained is that all the changes are visible to the caller*. In other words, the resulting execution semantics is as if both the caller and the callee were executing within the same address space. With NRMI or distributed call-by-reference (through remote pointers, as in Figure 2-3) this is done automatically. For call-by-copy, the programmer needs to simulate it by hand.

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

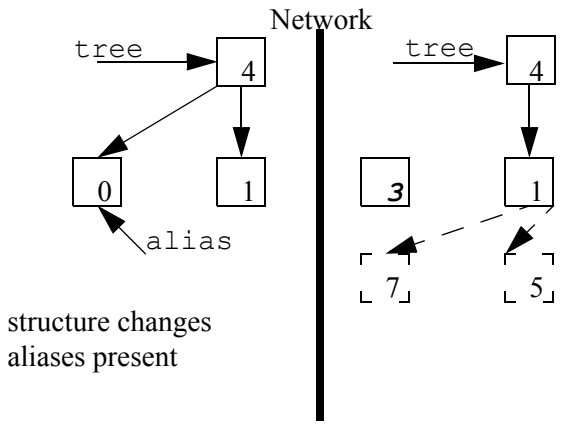
- In the first benchmark scenario, we assume that none of the objects reachable from the parameter is aliased by the client.



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



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



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

- In the first case, the parameter just has to be returned as the return value of the remote method. Once the remote call completes, the reference pointing to the original data structure gets reassigned to point to the return value. This will work for any changes to both the data and structure of the tree. The only complication here is that the method might already have a return value. Resolving this problem would require defining a special return class type that would contain both the original return type and the parameter. Besides the code for this new return class type itself, some additional code has to be written to call its constructor, populate it with its constituent members on the callee site, and retrieve them when the call completes.

- In the second case, the client needs to reassign the aliases pointing to some nodes in the original tree to point to the corresponding nodes in the new tree. After this step is performed, the reference reassignment described in the previous benchmark can be used. If

the programmer knows all the aliases, as well as where in the tree they point to (i.e., how to get to the aliased node from the tree root) then the aliases can be reassigned directly. If, however, the programmer only knows the aliases but not how to get to the aliased nodes, then a search needs to be performed before the update takes place. Both the original and the modified trees (which are now isomorphic) can be traversed simultaneously. Upon encountering each node, all aliases should be reassigned from pointing from the original tree to the modified one.

- In the third case, returning the changed structure alone is not very useful since the original and the modified trees are no longer isomorphic. To complicate matters further, the remote method invocation might make some changes to some of the tree nodes' data that were aliased by the caller and then disconnect them from the tree structure. This way the modified data nodes might no longer be part of the tree structure. Obviously just returning the new tree is not enough. Emulating the call-by-copy-restore or call-by-reference semantics is particularly cumbersome in this case. The simplest way to do it is by having the remote method create a "shadow tree" of its tree parameter prior to making any changes to it. The "shadow tree" points to the original tree's data and serves as a reminder of the structure of the original tree. Then both the parameter tree and the "shadow tree" are returned to the caller. The "shadow tree" is isomorphic to the original parameter and can be used for simultaneous traversal and copying of aliases. After that the new tree is used for the reference reassignment operation as in the previous cases. Note that correct update is not possible without modifying both the server and the client.

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

and third benchmark scenario, an extra 16 lines of code were needed to perform the updating traversal. For the third benchmark scenario, about 35 more lines of code were needed for the "shadow tree".

5.5.3 Experimental Results

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

The results of our experiments are shown in Table 5-1 to Table 5-6. All numbers are in milliseconds per remote call, rounded to the nearest millisecond. We ensured (by "warm-

ing" the JVM) that all measured programs had been dynamically compiled by the JVM before the measurements.

Table 5-1. Baseline 1—Local Execution (processing overhead) on both the fast (750MHz) and the slow (440MHz) machine.

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

Table 5-2. Baseline 2—RMl Execution, without Restore (one-way traffic).

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

Table 5-3. Baseline 3—RMI Execution with Restore on local machine (no network overhead).

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

Table 5-4. RMI Execution with Restore (two-way traffic).

Bench mark/ Tree Size	JDK 1.3				JDK 1.4			
	16	64	256	1024	16	64	256	1024
I	5	11	29	102	4	6	18	68
II	5	12	32	112	4	7	21	77

Benchmark/ Tree Size	JDK 1.3				JDK 1.4			
	16	**64**	**256**	**1024**	**16**	**64**	**256**	**1024**
III	6	13	38	143	4	9	27	106

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

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

Table 5-6. Call-by-Reference with Remote References (RMI).

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

The local measurements of Table 5-1 are given for both the fast and the slow machines. The local measurements of Table 5-6 are from the dual processor SunBlade machine. (This allowed us to avoid context switching and get a fair measurement. The numbers were significantly tainted on a uniprocessor machine.)

The main observations from these measurements are as follows:

- The benchmarks have very low computation times—their execution consists mostly of middleware processing and data transmission.

- Java RMI in JDK 1.4 is significantly faster than RMI in JDK 1.3. The speedup is in the order of 50-60% for this experimental setting. The speedup will be lower for a network that is slower relative to the processor speeds.

- The results of Table 5-4 minus the corresponding results of Table 5-3 will only yield an upper bound for the raw data transmission time, because the Table 5-3 results were computed exclusively on the fast (750MHz) machine while the Table 5-4 results include computation on both the fast and the slow (440MHz) node. The difference between the raw data transmission time and the "Table 5-4-minus-Table 5-3" value can be as high as the difference between the computation times on the fast and slow machines, shown in Table 5-1. Even then, however, JDK 1.3 seems to perform much better when no network is involved than the corresponding difference in JDK 1.4. This leads us to conclude that probably RMI in JDK 1.4 uses the underlying OS/networking facilities much more efficiently than JDK 1.3 and this difference disappears when the two hosts are sharing memory. We independently corroborated the raw data transmission time shown in the tables by profiling the benchmarks and noting the amount of time they spent blocked for I/O. We found that the real transmission time for JDK 1.3 is much higher even for transmitting the exact same amount of data as 1.4.

- For benchmarks I and II, NRMI is quite efficient. Even the portable version is rarely more than 30% slower than the corresponding RMI version. The optimized implementation of NRMI is about 20% slower than RMI in JDK 1.4. This is certainly fast enough for use in real applications. For instance, the optimized implementation of NRMI for JDK 1.4 is 20-30% faster than regular RMI in JDK 1.3.

- For benchmark III, which is hard to simulate by hand with call-by-copy alone, the portable implementation of NRMI gets similar performance to regular RMI in all cases, while the optimized implementation is faster. The cause is *not* the processing time for restoring the values changed by the header. (In fact, we performed the same experiments ignoring the manual restoring of changes and got almost identical timings.) Instead, the reason is that the regular RMI version transfers more data: the "shadow tree" is a simple way to

emulate the local semantics by hand, but stores more information than that of the NRMI linear map. (Specifically, it stores all the original structure of the tree instead of just pointers to all the reachable nodes.) The only alternative would be to compute a linear mapping to all reachable nodes on both sides, effectively imitating NRMI in user space.

- Call-by-reference implemented by remote pointers is extremely inefficient (as expected). Every access to parameter data by the remote method results in network traffic. Java RMI does not seem fit for this kind of communication at all—the memory consumption of the benchmarks grew uncontrollably. For the 1,024 node trees, the benchmarks took more than 600ms per case (repeated over 1,000 times) and in fact failed to complete as they exceeded the 1GB heap limit that we had set for our Java virtual machines. The reason for the memory leak is that the references back from the server to the client create distributed circular garbage. Since RMI only supports reference counting garbage collection, it cannot reclaim the garbage data.

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

5.6 The GOTECH Case Study

In this section we present an example of applying the GOTECH framework to convert a scientific application into a distributed application interacting with an application server. The original application is a thermal plate simulator. Its back-end engine performs

the CPU-intensive computations and its front-end GUI visualizes the results. (The back-end engine can also be configured to receive input from real heat sensors.)

The distribution scenario we want to accomplish is to separate the back-end simulation functionality from the rest of the application. and to place it on a powerful remote server machine. There are several benefits gained by this distribution scheme. First, it takes advantage of the superior computing power of a remote server machine. Second, multiple clients can share the same simulation server. Finally, if real heat sensors are used, the user does not have to be in the same physical location with the sensors to run the experiment.

The kind of distribution we examine is very similar to the distribution scenario of the Health Watcher application by Soares et al. [72]. (We sought to replicate the experiment of Soares et al. and re-engineer the Health Watcher system, but unfortunately the code is proprietary and was not made available to us.) The distribution scenario for Health Watcher was one where the GUI was running remotely from the core application and used a facade class to communicate with it. Near-identical issues are raised with our thermal plate simulator. Note, however, that, unlike Soares et al., we concentrate only on distribution and do not concern ourselves with persistence aspects.

A simplified UML diagram for the original version of the thermal plate simulator is shown in Figure 5-1. We have laid out the class diagram so that the front-end and back-end are clearly visible. The hierarchy under interface `Plate` contains the types of the objects that form the connecting link between the application's front-end and back-end. The graphical front-end creates a `Plate` object and several visual component objects reference it and query it to obtain the necessary data when performing their drawing operations. The `Plate`

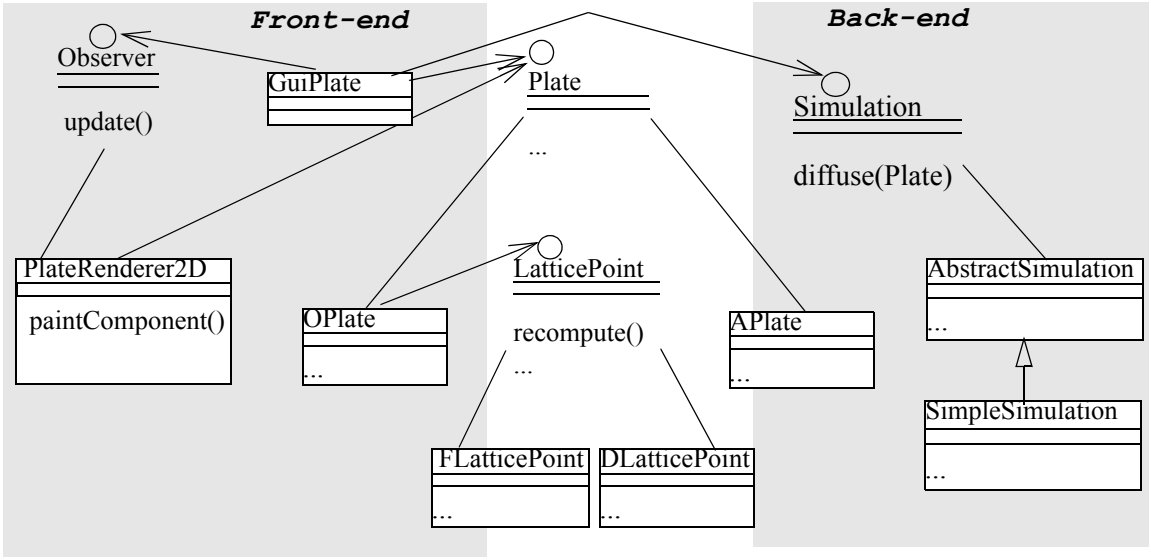


Figure 5-1: UML class diagram of the Thermal Plate Simulator functionality

object gets modified by being sent as a parameter to the `diffuse` method in the `Simulation` class. Once the `diffuse` method returns having modified its `Plate` parameter, the front-end is signaled to repaint itself. The visual components can access the updated data of the `Plate` object and redraw. Note that the main computation logic of the thermal plate simulation is not distributed—the results are the only data transferred over the network for remote display and simulation control.

Accomplishing the outlined distribution takes two steps:

- Converting simulation classes into EJBs and deploying them in an application server.

- Changing the client code to interact with an application server and EJBs instead of plain Java objects.

Notice that making simulation classes remote while preserving the original execution semantics requires special handling for remote method parameters. The `Plate` object that participates in a complicated aliasing (i.e. multiple referencing) scenario now becomes

a parameter of a remote call to an EJB. If a copy-restore mechanism is not provided by the application server, then the process of bridging the differences between local (by-reference) and remote (by-copy) parameter passing semantics becomes a tedious and complicated task. The use of NRMI (copy-restore semantics) completely eliminates the need for special purpose code to reproduce the back-end changes to the `Plate` object inside the front-end.

In-order for GOTECH to perform the required changes, we add some XDoclet-specific tags. Below are all the tags that are needed to convert a plain class `lattice.SimpleSimulation` into a stateless session Enterprise Java Bean.

```
/**
 * @ejb:bean name="SimpleSimulation"
 *           display-name="SimpleSimulation Bean"
 *           type="Stateless"
 *           transaction-type="Container"
 *           jndi-name="ejb/test/SSim"
 */

package lattice;
class SimpleSimulation {
...
/**
 * @ejb:interface-method view-type="remote"
 * @jboss:method-parameters copy-restore="true"
 */
 public void diffuse (Plate plate) { ... }
...
}
```

The tags entered in `lattice.SimpleSimulation` will convert the class into an EJB and will also change all its clients consistently. XDoclet generates the home and remote interface, as well as the bean class, all derived from the original source code for

`SimpleSimulation`. For example, the generated code for the home interface of the `SimpleSimulation` EJB (slightly simplified) is:

```
package simulations;
// [Redundant import statements removed]
/**
 * Home interface for SimpleSimulation.
 * @xdoclet-generated at [date] [time]
 */

public interface SimpleSimulationHome
  extends javax.ejb.EJBHome
{
 public static final String COMP_NAME =
   "java:comp/env/ejb/SimpleSimulation";
 public static final String JNDI_NAME =
   "ejb/SimpleSimulation";

 public simulations.SimpleSimulation create()
                        throws javax.ejb.CreateException,
                               java.rmi.RemoteException;
}
```

XDoclet also generates the non-code artifacts (deployment descriptor in XML) and an aspect that is supplied to AspectJ. AspectJ performs the client modifications based on the generated aspect. Recall how the aspect code generated by the template of Figure 3-1 will change all object creation (`new SimpleSimulation()`) to calls to a remote object factory and all method calls (e.g. `sim.diffuse(plate);`) to calls to a remote interface.

Upon completion, GOTECH has generated a new EJB, deployed it in the application server, and modified the client code to interact with the new bean. The new distributed application can be used right away without requiring any additional configuration.

5.7 J-Orchestra Case Studies

To showcase J-Orchestra, we present four case studies of partitioning medium to large applications and of several smaller applications. The first three case studies demonstrate the benefits of appletizing by successfully transforming three realistic, third-party applications: JBits [27], JNotepad [36], and Jarminator [35], into client-server applications. JBits is not only the largest among all the case studies but also a commercial application available in bytecode-only form. While JNotepad and Jarminator are also third-party applications, they are free and publicly available. The last case study of partitioning the Kimura system [55] demonstrates how automatic partitioning can be an enabling technology for prototyping ubiquitous computing applications [97]. In this case study, the starting point is a distributed application that is rewritten with the explicit purpose to develop a centralized version that will later become distributed with J-Orchestra. Finally, we present some of the most representative smaller applications we have partitioned with J-Orchestra.

5.7.1 Appletizing Case Studies

In our measurements, we compare the partitioned applications' behavior to using a remote X display [71] to remotely control and monitor the application. Since all three subjects are interactive applications and we could not modify what they do, we got measurements of the data transferred and not the time taken to update the screen (i.e., we measured bandwidth consumption but not latency). Our experience is that appletizing is an even greater win in terms of perceived latency. In all cases, the overall responsiveness of the appletized versions is much better than using remote X displays. This is hardly surprising, as many GUI operations require no network transfer. Note that the data transfer numbers

would not change in a different measurement environment. For reference, however, our environment consisted of a SunBlade 1000 (dual UltraSparc III 750MHz, 2GB RAM) and a Pentium III, 600MHz laptop connected via 10Mbps ethernet.

5.7.1.1 JBits

JBits, the largest of the applications, is an FPGA simulator by Xilinx—a web search shows many instances of industrial use. The JBits GUI (see [27] for a picture of an older version) is very rich with a graphical area presenting the results of the simulation cells, as well as multiple smaller areas presenting the simulated components. The GUI allows connecting to various hardware boards and simulators and depicting them in a graphical form. It also allows stepping through a simulation offering multiple views of a hardware board, each of which can be zoomed in and out, scrolled, and so forth. The JBits GUI is quite representative of CAD tools in general.

JBits was given to us as a bytecode-only application. The installed distribution (with only Java binary code counted) consists of 1,920 application classes that have a combined size of 7,577 KBytes. These application classes also use a large part of the Java system libraries. We have no understanding of the internals of JBits, and only limited understanding of its user-level functionality.

For our partitioning, the vast majority (about 1,800) of the application's classes are anchored by choice on the server. Thus co-anchored objects can access each other directly and impose no overhead on the application's execution. This is particularly important in this case, as the main functionality of JBits is the simulation, which is compute-intensive. With the anchoring by choice, the simulation steps of JBits incur no measurable overhead.

259 classes are always anchored on the client (i.e., GUI) site. Of these, 144 are JBits application classes and the rest are classes from the Java system's graphical packages (AWT and Swing). The rest of the classes are anchored on the server site. (We later discuss a variation in which we make some objects mobile.)

The appletized JBits performs arbitrarily better than a remote X-Window display. For instance:

- JBits has multiple views of the simulation results ("State View", "Power View", "Core View", and "Routing Density View"). Switching between views is a completely local operation in the J-Orchestra partitioned version—no network transfers are caused. In contrast, the X window system needs to constantly refresh the graphics on screen. For cycling through all four views, X needed 3.4MBytes transferred over the network.

- JBits has deep drop-down menus (e.g., a 4-level deep menu under "Board->Connect"). Navigating these drop-down menus is a local operation for the J-Orchestra partitioned application, but not for remote access with the X window system. For interactively navigating 4 levels of drop-down menus, X transferred 1.8MBytes of data.

- GUI operations like resizing the virtual display, scrolling the simulated board, or zooming in and out (four of the ten buttons on the JBits main toolbar are for resizing operations) do not result in network traffic with the appletized JBits. In contrast, the remote X display produces heavy network traffic for such operations. With our example board, one action each of zooming-in completely and zooming-out results in 3.5MBytes of data transferred. Scrolling left once and down once produces about 2MBytes of data over the network with X, but no network traffic with the J-Orchestra partitioned version. Continuous scrolling over a 10Mbps link is unusably slow with the X window system. Clearly, a slower connection (e.g., DSL) is not suitable for remote interactive use of JBits with X.

Even for a regular board redraw, in which the appletized JBits needs to transfer data over the network, less data get transferred than in the X version. Specifically, the appletized version needs to transfer about 1.28MB of data for a complete simulation step including a redraw of the view. The X window system transfers about 1.68MBytes for the same task. Furthermore, J-Orchestra transfers these data using five times fewer total TCP segments, suggesting that, for a network in which latency is the bottleneck, X would be even less efficient.

Although there may be ways (e.g., compression, or a more efficient protocol) to reduce the amount of data transferred by X, the important point is that some data transfer needs to take place anyway. In contrast, the appletized version only needs to transfer a data object to the remote site, and all GUI operations presenting the same data can then be performed locally. For the cases that do produce network traffic, the appletized version can also have its bandwidth requirements optimized by using a version of Java RMI with compression.

Experiment: Mobility

In the previous discussion we did not examine the effects of object mobility. In fact, very few of the potentially mobile objects in JBits actually need to move in an interesting way. The one exception is JBits View Adaptor objects (instances of four `*ViewAdaptor` classes). View adaptors seem to be logical representations of visual components and they also handle different kinds of user events such as mouse movements. During our profiling we noticed that such objects are used both on the server and the client partition and in fact can be seen as carriers of data among the two partitions. Thus, no static placement of all view adaptor objects is optimal—the objects need to move to exploit locality. We specified

a mobility policy that originally creates view adaptors on the client site, moves them to the server site when they need to be updated, and then moves them back to the client site.

Surprisingly, object mobility results in more data transferred over the network. With mobile view adaptor objects and an otherwise indistinguishable partitioning, J-Orchestra transferred more than 2.59MBytes per simulation step (as opposed to 1.28MBytes without a mobility policy). The reason is that the mobile objects are quite large (in the order of 300-400KBytes) but only a small part of their data are read/written. In terms of bytes transferred it would make sense to leave these objects on one site and send them their method parameters remotely. Nevertheless, mobility results in a decrease in the total number of remote calls: 386 remote calls take place instead of 484 for a static partitioning, in order to start JBits, load a file and perform 5 simulation steps. Thus, the partitioned version of JBits with mobile objects may perform better for high bandwidth networks, in which latency is the bottleneck.

5.7.1.2 JNotepad

JNotepad emulates the functionality of the Windows Notepad editor. It allows the user to read and write text files. As in any simple text editor, the functionality of JNotepad consists of a user interface and I/O facilities. The user manipulates the content of a text file through the user interface, which includes the interaction with the I/O facilities for writing and retrieval of files to and from disk. One appletizing scenario for Notepad places the user interface on the client, while processing the I/O on the server.

The analysis for appletizing showed that the application has a total of 106 classes (66 JRE system classes, and 40 application classes). It also assigned 98 classes to the client site,

7 classes to the server site, and left 2 classes unassigned. To help determine a good placement for the unassigned classes named `Center` and `Actions`, we performed a scenario-based profiling that consisted of opening a file, searching for a word in it, changing its content, and saving it back to disk. The data exchange patterns, revealed by the profiling, showed that the `Center` class has been tightly coupled with the client classes, calling each other's methods 17 times. Therefore, the most logical placement for this class is on the client, together with the GUI classes. The `Actions` class exhibited a more complex data exchange pattern, communicating with both the client (18 method calls) and the server (42 method calls). More detailed profiling showed that the data exchange between the server classes and the `Actions` class happens inside the `savE` method, with the rest of the methods communicating only with the client classes. This is exactly a case for which object mobility can provide an elegant solution. The objects of type `Actions` can be created at the client site and then temporarily move to the server for the duration of the `savE` method. As our measurements have shown, this mobility arrangement does not result in less data being transferred over the network, but significantly decreases the number of remote calls made (from 60 to 17).

We compared the behaviors of the partitioned application to the original one, run remotely under the X window system. The test scenario was similar to the profiling one, described above. (We believe that this reflects typical JNotepad use.) The appletized version transferred less than 1/7th the amount of data over the network (~1 MB vs. ~7 MB). With all the GUI operation not generating any network traffic, the appletized version sent data over the network only when reading and writing the text file. Under X, JNotepad, run-

ning on the server that had the text file, accessed it directly. However, its every interaction with the GUI resulted in sending data over the network.

5.7.1.3 Jarminator

Jarminator is a popular Java application that examines the content of multiple jar files and displays their combined content in a tree view. The user can have only a subset of the content displayed by supplying a wildcard filter. We have appletized Jarminator so that it can examine jar files on a remote machine and display the results locally. The analysis for appletizing showed that the application uses a total of 74 classes: 55 JRE system classes, and 19 application classes. The appletizing analysis assigned 62 classes to the client site, 4 classes to the server site, and left 8 classes unassigned. A case-based profiling suggested assigning 6 classes to the client, 1 to the server, and did not detect any data exchange with the remaining class. It also did not reveal any communication patterns in which a mobility scenario could be useful.

Again, we compared the behaviors of the partitioned application to the original one, run remotely under the X window system. In this benchmark, we used Jarminator to explore three third-party jar files used by J-Orchestra. The use scenario included loading the jars, navigating through the tree view, and applying wildcard filters to the displayed content. The appletized version exhibits significant benefits, transferring less than 1/30th the amount of data over the network (~500 KB vs. ~15 MB). In fact, operations such as filtering the displayed contents are entirely local in the appletized version and do not generate any network traffic.

5.7.1.4 Discussion

Appletizing, just like general application partitioning, is not free of limitations. Applications can be arbitrarily complex and can defy correct partitioning. Furthermore, although we handle common cases of invalid operations inside applets, we do not have an exhaustive approach to sanitize all Java code for applet execution. More common in practice, however, is the case of applications that can be correctly appletized (i.e., they do not employ unsupported Java features such as dynamic loading or code rejected by the applet security manager) yet require manual intervention to override conservative decisions of the J-Orchestra heuristic analyses.

Of our three case studies, JNotepad and Jarminator were partitioned completely automatically within 1-2 hours of time. JBits required more intervention (but still no explicit programming) to arrive at a good partitioning within 1-2 days. For example, knowing only the JBits execution from the user perspective, we speculated that the integer arrays transferred from the server towards the GUI part of JBits could safely be passed by-copy. These arrays turned out to never be modified at the GUI part of the application. A more conservative rewrite would have introduced a substantial overhead to all array operations.

Even in the less automatic cases, however, the expertise required to appletize an application is analogous to that of a system administrator, rather than that of a distributed systems programmer. For instance, in the JBits case we partitioned a 7.5MB binary application without knowledge of its internals. Even though the partitioning was not automatic, the effort expended was certainly much less than that of a developer who would need to change an application with about 2,000 classes, more than 200 of which need to be modified to be accessed remotely.

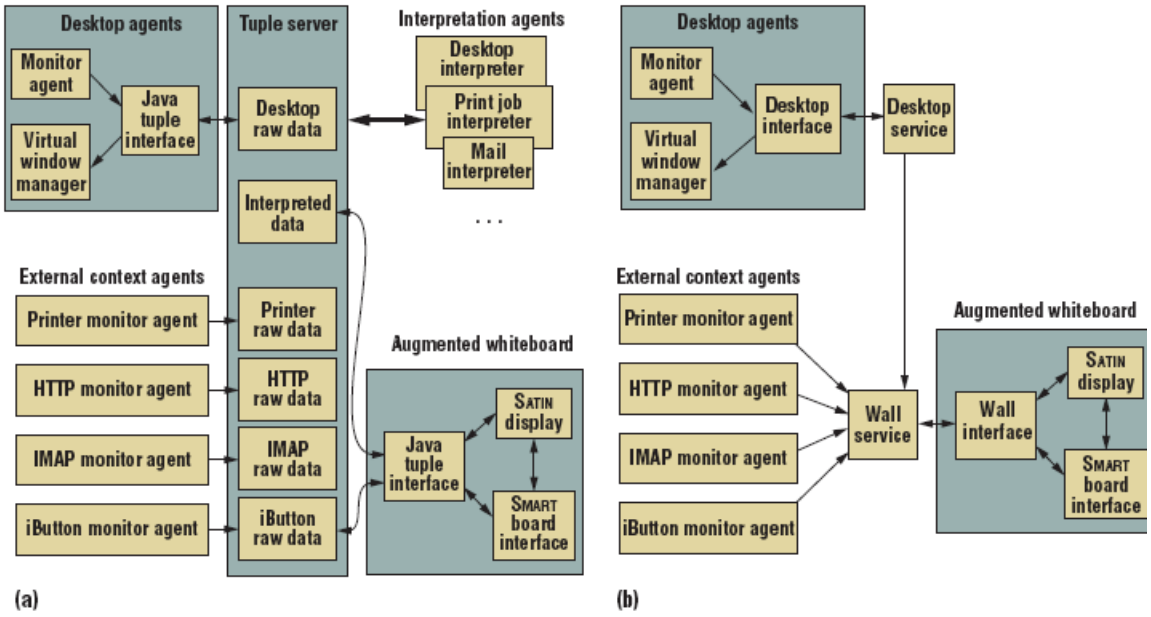


Figure 5-2: Kimura architecture: (a) the original system; (b) the reengineered Kimura2 system.

Our experience confirms the benefits of appletizing. Indeed, it requires no programming: we did not have to write distribution code or recode the subject applications; it is flexible: each of the subjects has a complex GUI and could not be written as a servlet; it is easy to deploy: all subjects run as applets over a standard browser communicating with a server part; and results in good performance: by putting the GUI code on the client, we transmit less data than transferring all the graphics.

5.7.2 Kimura Case Study

The Kimura case study [51] stands apart from other J-Orchestra case studies because its primary objective was not only to showcase the capabilities of J-Orchestra but also to explore whether automatic application partitioning can help researchers rapidly prototype distributed ubiquitous computing systems. Proponents of ubiquitous computing (or ubicomp, for short) [97] envision a future in which computers are inexpensive and plentiful

and seamlessly interoperate. Ubicom is also one area in which researchers have clearly identified the need for software engineering support [1]: although hardware continues to become smaller and cheaper, the corresponding software tools that would make the vision of ubicomp possible have not matured at the same rate. One major feature of the ubicomp domain—distinguishing it from traditional desktop applications, for example—is the software's inherent distributed nature. Ubicomp environments are naturally distributed over multiple computers connected via a wired or wireless network. These computers come in many shapes and sizes, from handheld to wall-sized. Applications in these environments are typically designed under the assumption that computing resources come and go in ever-changing combinations of lightweight and heavyweight, predefined and ad hoc groups. So, ubicomp application developers typically must suffer all the complexities of distributed-systems programming.

The difficulties that developers encounter when building ubicomp applications are more pronounced during research and prototype development. Ubicomp application prototypes are typically exploratory: The application's structure, the kind of data being shared, and the data's distribution characteristics will change frequently as the application undergoes iterations through the design-build-deploy-evaluate-redesign cycle. To facilitate application prototyping in this domain, developers must be able to modify the data structures' underlying distribution characteristics with little effort. Unfortunately, ubicomp developers often aren't expert at distributed systems. As a result, ubicomp researchers need simple, automated techniques that support rapid prototyping in such domains, and the Kimura case study explores how useful automated application partitioning with J-Orchestra can be in this respect.

As a larger case study of applying automatic partitioning to ubiquitous computing systems, we used automatic partitioning in developing the latest version of the Kimura system, which is a realistic, complex ubicomp application [55], Kimura is part of a research project that seeks to explore and evaluate the addition of visual peripheral displays to human-computer interfaces. Kimura uses large, projected displays as peripheral interfaces to complement an existing work area—the area surrounding a traditional desktop computer. Kimura uses these peripheral displays to help users manage multiple activities, such as coherent sets of tasks typically involving multiple documents, tools, or communications. Kimura assists in visualizing background activities as montages of images on the peripheral displays. These montages serve as anchors for background awareness information collected from a context-aware infrastructure.

Kimura's source code consists of 98 Java application classes and over 4,400 source statements. These application classes use many system and third-party classes, including Swing and Java Advanced Imaging (JAI) library classes, as well as classes that facilitate two-way communication with an electronic whiteboard.

The architecture of the original version of Kimura consists of three distinct components (see Figure 5-2a). A desktop interface module runs on the user's PC, monitoring all window and application activity through a native library and providing virtual-desktop functionality. A context interpreter module acts as an intermediate layer, aggregating the incoming messages from the desktop and the context-aware infrastructure and conveying them to the peripheral-display module, which we informally call "the wall." The wall, which connects directly to several projectors and a SMART Board interactive whiteboard,

maintains two-way communication with the SMART Board and provides up-to-date visualizations of the user's working contexts as montages projected onto the SMART Board.

These three components connect through TSpaces, a communication package designed to connect distinct distributed components [49]. TSpaces is based on the well-known tuplespace paradigm and incorporates database features such as transactions, persistent data, and flexible queries. It employs the publish-subscribe model. When one component adds or deletes a tuple on the TSpaces server, an appropriate callback method is called asynchronously in any other component that has registered to receive notifications matching that type of tuple.

The creators of TSpaces aimed at "hitting the distributed computing sweet spot [49]." The system lets programmers ignore many hard aspects of distributed communication, such as naming, state, and load balancing. The original Kimura implementation didn't use any of these advanced TSpaces features but employed TSpaces as a convenient way to keep shared state and to broadcast global events—such as activity changes—to all system components.

To evaluate the applicability of automatic partitioning to the ubicomp domain, we reengineered Kimura by removing the existing distribution code and redistributing it with automatic partitioning. The first step of reengineering was to separate Kimura's main application tasks from its network communication. In this way, we could create a simpler Kimura core, evolve it as necessary, and automatically partition it with J-Orchestra.

We first removed the code that supported distribution with TSpaces and replaced it with a single shared data structure. The result was a single program that could run in one process and open multiple windows—the wall and the desktop control panel—on a single

machine. The TSpaces-related code—functions responsible for connecting to the TSpaces server and adding or deleting tuples—was spread over 11 of the 77 source files. While TSpaces dictated an event-based structure for the application, the centralized version could use direct method calls between components, resulting in simpler, cleaner code.

Similarly, the interpreter component, which acted as an extra level of indirection between the desktop and the wall, was superfluous in the centralized version. We removed it as a distinct entity, preserving its functionality in two new modules designed to act as public interfaces of the desktop and the wall. These two new modules were two singleton classes whose responsibilities included handling incoming and outgoing messages from the application's other part. Coding and integrating them with the rest of the system was straightforward. As Figure 5-2b illustrates, Kimura's new version no longer has a central server. Instead, the system components talk to one another directly and synchronously.

Kimura2 consists of two partitions—one for the desktop and one for the peripheral display. The user interaction takes place through the peripheral display, while the desktop machine does the core of the processing, such as monitoring open applications. One can think of the peripheral display as a "monitoring console" for the Kimura working environment.

To make partitioning possible, we had to understand the application's internal structure, as the type based analysis heuristics of J-Orchestra, which determines what references can leak to what code, turned out to be too conservative in this case. Kimura2 uses Swing UI classes on what would become the wall and the desktop partitions. Because the code handling these objects is unmodifiable, we need to be sure that the objects in one partition are not shared in the other. Otherwise, the Swing code might try to access a remote object's

fields directly, resulting in a crash. The heuristic analysis conservatively concluded that Swing classes can't exist on two different partitions. However, we know that the Swing object partitioning in Kimura2 is safe: the Swing widgets on the desktop display are distinct from the Swing widgets on the wall display. Therefore, we could explicitly direct J-Orchestra to produce appropriate code for Swing classes on both partitions.

Altogether, of the 64 automatically rewritten classes, 43 were Swing and Abstract Window Toolkit (AWT) classes, and 6 were made serializable so that they could be by-copy passed by-copy across different memory spaces to improve performance. We excluded 71 Kimura application, 4 third-party, and 12 Java development kit (JDK) classes from the distribution process altogether because we determined that they never participate in the distributed communication. All in all, including testing, it took us only a few days to partition Kimura2 with J-Orchestra.

Discussion

Automatic partitioning turned out to be quite beneficial in the case of developing Kimura2. The main benefit is in the new software architecture's simplicity, which resulted in more understandable and maintainable code without sacrificing any of the original functionality. Kimura2's architecture will facilitate planned additions to the system much more easily because the developers can focus on the desired functionality without worrying about the distribution specifics. The new version also is easier to deploy because we don't need to maintain a running TSpaces server.

Table 5-7. Software Complexity Metrics

	Kimura	Kimura2	Percent more in original
Total statements	4,436	4,084	8.6
Number of classes	98	92	6.5
Number of methods	693	682	1.6
Program difficulty metric	3,305	3,124	5.8
Development effort metric	2,611	2,235	16.8
Lack of cohesion of methods metric	2,395	2,165	10.6
Interpackage fan-out (no. of classes)	881	822	7.2

To quantify this simplicity's benefits, we used JStyle [38] to derive software metrics. The software engineering community is still divided on software metrics' value and meaning, so the significance of our qualitative findings is somewhat subject to individual interpretation. Table 5-7 lists some of the more pronounced differences between Kimura and Kimura2. The new version exhibited better results in all metrics, including those not described in detail here.

Kimura originally consisted of 4,436 source statements (including declarations but not counting comments, empty statements, empty blocks, closing brackets, or method signatures). Out of them, 3,836 (86 percent of the total) remained unchanged in the new version. We completely removed the TSpaces-related code (486 statements, almost 11 percent of the total) and added 134 statements. Finally, we modified 114 statements to adapt the application to the new communication paradigm.

As Table 5-7 shows, the new version exhibited significant differences using the Halstead program difficulty metric [28], Chidamber and Kemerer's lack of cohesion of methods (LCOM) [16], and class fan-out (the number of classes a given class depends on). The new version is significantly less complex. Of course, we would expect a centralized architecture to be much less complex than a distributed one. However, it is interesting to quantify the difference.

In our evaluation of Kimura2, we also performed extensive measurements to evaluate how the partitioning infrastructure affects performance. Most system operations (including montage creation, montage switching, and document manipulation) exhibited significant speedup in relation to their counterparts in the original version, with only two of the measured operations (wall montage switching and document activation) showing a slowdown. We omitted our performance measurements because they are not essential to our conceptual evaluation of automatic partitioning for ubicomp, being merely the result of orthogonal, low-level concerns, such as the underlying middleware used in the case of J-Orchestra relative to TSpaces.

Our experiences using automatic partitioning to develop ubicomp applications have been quite positive. The approach's overwhelming advantages include both the simplicity of coding for a single machine without the need for distributed programming and the ease of repartitioning and redeployment. Furthermore, the ability to run on unmodified runtime systems—that is, any Java VM—is invaluable when using a multitude of heterogeneous devices. Nevertheless, we have also identified several shortcomings associated with automatic partitioning for ubicomp applications but not necessarily revealed by the Kimura case study. Most of these shortcomings stem from the fact that many general engineering

issues are difficult to address using an automated approach that J-Orchestra follows. Contrary to this automated approach, which involves no programming, just resource-location assignment (e.g., graphics code should run on this machine, or the main engine should run on that machine), a semiautomatic approach could let the user annotate detailed parts of the code and data that would actuate advanced distributed systems mechanisms (e.g., what data should be replicated, how the copies should remain consistent, and where leases should be used for fault tolerance). Indeed, a semiautomatic approach could resolve many of the issues associated with automatic partitioning, and we discuss this research direction in the future work section (Chapter VIII).

5.7.3 Other J-Orchestra Case Studies

Some of the most representative other applications we have partitioned to demonstrate J-Orchestra include:

- the Java Speech API demo mentioned in Section 4.2 on page 63. Speech is produced on one machine while the application GUI is running on a handheld (IPaq machine). In general, Java sound APIs can easily be separated from an application's logic using J-Orchestra.

- JShell: a third-party command shell for Java. The command parsing is done on one machine, while the commands are executed on another.

- PowerPoint controller: we have written a small Java GUI application that controls MS PowerPoint through its COM interface. We partitioned the GUI of this application from its back-end. We run the GUI on a IPaq PDA with a wireless card and use it to control a Windows laptop. We have given multiple presentations using this tool.

- A remote load monitoring application: machine load statistics are collected and filtered locally with all the results forwarded to a handheld (IPaq) machine over a wireless connection and displayed graphically. The original application was written to run on a single Windows machine.

This chapter discussed the issues of applicability of the three software tools for separating distribution concerns explored by this dissertation. We identified programming scenarios under which NRMI, GOTECH, and J-Orchestra would be most useful. Finally, we presented case studies that showcased how the tools can successfully separate distribution concerns of realistic applications.

CHAPTER VI

GENERALIZING THE J-ORCHESTRA INDIRECTION MACHINERY

This chapter discusses how one of the technical contributions of this dissertation can be generalized to domains other than distributed computing. Specifically, we take a closer and broader look at the J-Orchestra approach to enabling indirection in the presence of unmodifiable code (e.g., Java system classes), which is one of its sources of scalability. Chapter IV discussed the J-Orchestra analysis heuristic that determines which application objects get passed to which parts of native (i.e., platform-specific) code and a technique for injecting code that will convert objects to the right representation so that they can be accessed correctly inside both application and native code. Here we discuss the broader ramifications and limitations of user-level indirection and show how the approach taken by J-Orchestra can be fine-tuned so that user-level indirection can be applied to more system classes.

6.1 Introduction

In this chapter, we take a more general look at user-level indirection techniques and show that all different versions of the idea converge into using the same general approaches. Then we discuss why the presence of native code always results in correctness limitations. Some of these limitations are straightforward (e.g., native code can have its own state) while some others are more subtle (e.g., native code can change user-level state

directly). Despite the fact that we generally use Java (i.e., Java language syntax, Java terminology, and JNI conventions) as our reference system, our observations apply to most other runtime systems for platform-independent binary code applications such as the CLR and .NET technologies.

6.2 User-Level Indirection Techniques

We use the name "user-level indirection" to describe any general technique that transparently interposes extra functionality to the execution of existing applications by using code transformation techniques, instead of modifying the underlying implementation of the runtime system. Applications of user-level indirection include transparent distributed execution [21][66][74][75][84], persistence [12][46][59], profiling [33], and logging [48]. In general, user-level indirection aims at capturing specific events and performing actions whenever they occur. Such events typically are:

- Access to a field of an object or a static field (reading or modifying the field).
- Calls to a method of an object of a specific type, or calls to a static method.
- Object construction.

For instance, we may want to add indirection to all changes to the fields of an object for logging: we may want a permanent log of all state updates in a running system. This is possible by finding all field access instructions in the application and modifying them to log their action before taking it. The logging code is either included inline at the field access site, or a separate method can be called.

What complicates user-level indirection is the existence of reusable core functionality in the form of *system classes* (a.k.a. *standard library classes*). User-level indirection

cannot afford to ignore system classes, *even if the intended use is not concerned with system-level events*. For instance, consider a user-level indirection system that performs actions every time a user-level method gets called. User-level methods, however, often get called by system-level code. For instance, system libraries often accept a callback object and invoke its methods in response to asynchronous events, or in response to system code actions initiated by a user-level call. Thus, the user-level indirection technique needs to ensure that it allows and correctly handles all calls, regardless of whether they occur inside user-level or system-level code.

In popular modern runtime systems, the majority of system class code is not special. Most of the Java system classes, for instance, are distributed in Java bytecode format. Thus, one can apply the same user-level indirection techniques to both user-level code and byte-code-only system classes. Indeed, several systems [22][84] follow this approach. The standard technique in this case is to create a separate, instrumented version of the system classes. The instrumented version co-exists with the standard system classes in the same application. In this way, an application can access both the user-level indirected versions of system classes and the original versions without any conflict. This is necessary, since the system classes are often used inside the instrumentation code itself. In original application code, however, all uses of system classes are replaced with uses of their instrumented counterparts. Reference [22] calls this the "Twin Class Hierarchy" approach (TCH). As an example, imagine that the original Java application contains code such as:

```
class A {
  public java.lang.String meth(int i, B b) {...}
}
```

The rewritten class would use the instrumented class types:

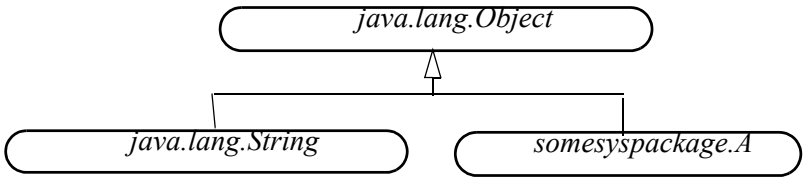


Figure 6-1: (a): Original system classes hierarchy

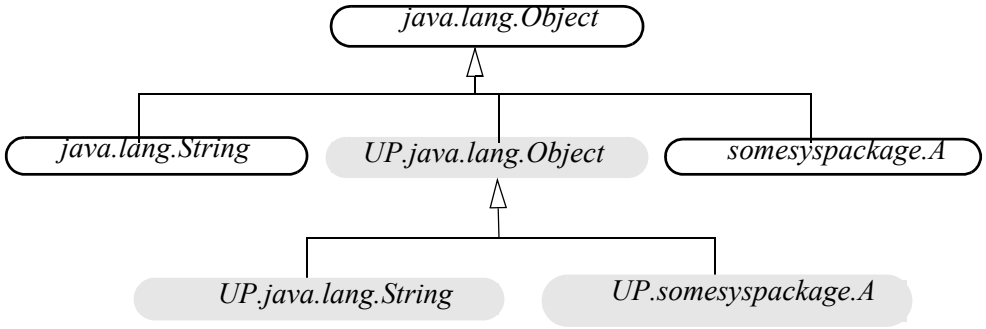


Figure 6-1: (b): Replicating system classes in a user package ("UP")

```
class UP.A {
  public UP.java.lang.String meth(int i, UP.B b)
  {...}
}
```

(`UP` in the above code stands for "user package".) Figure 6-1 shows the effects on the class hierarchies pictorially.

6.3 Transparency Limitations

The problems with any user-level indirection technique begin when a system class with native code needs to be instrumented. Native code (a.k.a. *platform-specific binary code*) is often used to implement system-level functionality. Some of the most fundamental system classes (e.g., the ones dealing with threading, file and network access, GUI, and so forth) rely on native code, mainly for reasons of low-level resource access, such as context-

switching or fast graphical operations. System classes with native code are, thus, a way to export runtime system functionality as language-level facilities.

Native code cannot be instrumented without invalidating all the advantages of the user-level indirection approach. Changing native code requires platform-specific changes and the creation of special versions of the runtime system (either the executable program or its dynamic libraries). Similarly, analyzing native code and relying on its implementation properties is a platform-specific task. Thus, dealing with native code is incompatible with the main motivation for user-level indirection: that of portability and platform independence. Therefore, native code is *opaque* for the purpose of user-level indirection: it can be neither modified nor analyzed.

Having an application access opaque code immediately introduces limitations in user-level indirection approaches. Even if opaque code is a small percentage of the total system code,[1] it is likely to be used by every application and needs to be handled correctly. (In fact, because `java.lang.Object` and `System.Object`, the root classes in Java and C#, respectively, use native code in their implementation, one could argue that every program written in these languages contains opaque code.) Clearly, one limitation is that user-level indirection cannot be used to intercept actions occurring entirely inside native code. For instance, we cannot observe and log updates to program state kept inside native code: such state is invisible to the user-level. That is, changes to internal system state (e.g., the contents of a low-level window, the scheduling structure of threads, and so forth) cannot be intercepted using user-level indirection. Although it may seem that such state is low-level

1. Only about 3% of the Java system classes have native methods. (All numbers were measured on Sun JDK 1.4.2.) Nevertheless, as we show later, these are some of the most commonly used classes in Java and are likely to constitute a much larger percentage of the loaded system code in a Java application.

and is outside the scope of user-level indirection, the restriction nevertheless places boundaries on what is achievable with user-level indirection alone. For instance, without reliance on implementation specifics of the Java system libraries, a distributed execution system that relies on user-level indirection (similarly to J-Orchestra: Pangaea [74], Addistant [84], and JavaSplit [21]) cannot hope to transparently migrate window or thread objects from one machine to another. This task can still be achieved by special-purpose emulation of the semantics of a thread or window at the user level, but not by employing general-purpose user-level indirection techniques on the Java system classes.

Often, however, the interactions of native code with user-level indirection are more subtle. In the Java system, native code can directly read or modify the state of object fields declared in bytecode. This allows for tight integration of native code and Java code. Essentially, the Java Native Interface (JNI) is a way to program using the full object model of the JVM with C or C++ as the host language. Direct access to fields inside native code complicates matters for user-level indirection. Consider the TCH user-level indirection approach for instrumenting standard Java libraries [22]. (This approach is representative of other user-level indirection techniques, including the one in J-Orchestra.) In this approach, if a class `A` has a native method, an instrumented version of `A` delegates calls to the native method of an internal `A` object. This technique is used because a native method implementation in Java is bound to a particular class name and cannot be reused for a different class. For instance, consider original code as follows: (This code does not reflect the Java `File` class but the structure is representative of several system classes with native methods.)

```
class File {
  ...
  public native void write(byte b);
}
```

The instrumented version of this class would be:

```
class UP.File {
  private File origImpl_;
  ...
  // delegate to native method
  public void write(byte b) {origImpl_.write(b);}
}
```

It may at first seem that the `UP.File` class can use arbitrary user-level indirection for its non-native methods. Nevertheless, this is not the case. Imagine that the `File` class also has a non-native method `newLine`:

```
class File {
  ...
  public native void write(byte b);
  public void newLine() { ... }
}
```

It is not safe to indirect method `newLine` (e.g., to track its changes to fields of a `File` object) yet simply delegate method `write`. To see this, consider the re-written code:

```
class UP.File {
  private File origImpl_;
  ...
  // delegate to native method
  public void write(byte b) {origImpl_.write(b);}
  public void newLine() {...} // instrumented body
}
```

The problem is that any call to method `write` affects the `origImpl_` object, while any call to method `newLine` affects the current object of type `UP.File`. Separating these two objects (when they were one in the original application) destroys the transparency of

user-level indirection. Therefore, we see that the TCH user-level indirection approach is all-or-nothing: any class that has even a single native method is impossible to instrument transparently. This limitation is not specific to the TCH approach: following the same reasoning one can see that once a class has native methods, it is not possible to transparently replace it with an instrumented copy of the class such that it implements any kind of user-level indirection.

The ability of Java native system code to directly access user-level state hinders many more user-level indirection tasks. For instance, consider user-level indirection approaches that capture all updates to fields of an object (e.g., to implement transparent persistence or distributed execution). In this case, all objects that can ever be referenced by native code cannot be fully indirected using user-level indirection techniques. That is, even if an object's class has no native methods, if the object is ever referenced by some other class's native code, then we cannot indirect all access to the object's fields.

Furthermore, often constraints on the use of user-level indirection have to do with restrictions derived from the structure of the user-level indirection scheme itself. For instance, consider again the above TCH rewrite. Without any special provisions, the limitations on the use of indirection propagate to all subclasses. A subclass `ROFile` of the original `File` class may have no native methods, yet its methods cannot be instrumented. If the instrumentation were performed, the `UP.ROFile` class would be a subclass of `UP.File` and not of `File`. Thus, `UP.ROFile` would not be able to access non-public members of `File`. We later discuss how to remove this limitation.

6.3.1 Beyond Java Conventions: Native Code in .NET

For the purposes of our discussion, the .NET and Java technologies are almost equivalent, with .NET being slightly more restrictive due to the unstructured nature of interfacing between managed and unmanaged code. Just like in the Java case, managed and unmanaged code in the CLR can operate on the same objects. Just like in Java, .NET unmanaged code, usually written in C++, provides many system services that are impossible to implement in a managed environment because they require such low-level programming techniques as direct memory access. Unlike the Java platform, however, which clearly distinguishes between bytecode and native libraries and provides a clean interfacing mechanism between the two in the form of the JNI, the C# core classes implementation consists of managed and unmanaged code that are binary compatible with each other.

At the language level, the annotation `[MethodImplAttribute(MethodImplOptions.InternalCall)]` specifies external methods that are implemented natively in the runtime itself. These methods use standard Microsoft C language calling conventions (such as `__stdcall` and `__cdecl`). In addition, the internal member methods in C# take this as the first argument, which in C++ becomes just a regular pointer that can be used to access and modify the memory of the underlying C# class directly. For example, a brief look at the Microsoft Shared Source CLI Implementation reveals that the C++ native code of the runtime relies on a very concrete object memory layout. For example, comparing whether two C# references point to objects of the same type includes comparing the pointers to their method tables, located at a predefined memory offset from the base references. Therefore, unmanaged code in the CLR not only accesses fields of objects, but is allowed to make assumptions about how these fields are laid out in memory. Such tight coupling between

managed and unmanaged code enables an efficient implementation for the runtime but also makes introducing any indirection into the managed code almost impossible. Therefore, introducing indirection by simply moving code of a Core Library C# class with native dependencies to a different package is even more unrealistic and error-prone than it is in Java. In the remainder of this chapter, all our qualitative observations should apply equally well to the CLR, unless we explicitly note otherwise.

6.4 Weak Assumptions of J-Orchestra Classification

To determine which program actions can be safely indirected, we would need to analyze the implementation of native methods. Since source code for the VM and its dynamic libraries will typically not be available, one important question is whether one can use the type information at the native code interface as a "poor-man's native code annotations." We discuss how some well-founded assumptions on the behavior of native code enable J-Orchestra to employ a conservative type-based analysis of what objects can be accessed by native code. It turns out that type information is often remarkably sufficient for determining the safety of user-level indirection.

6.4.1 Type-Based Analysis + Weak Assumptions

Recall that the majority (~97%) of Java system classes have no native methods. Such classes encode useful reusable libraries and not system-level functionality. It is, thus, crucial to automatically recognize system classes that do not interact with native code and to support correct user-level indirection for them. In general, this task is impossible without making assumptions regarding native code behavior. For instance, all classes in Java are subclasses of the `java.lang.Object` class, which has native code. In theory, any native

method can be receiving an `Object`-typed argument, discovering its actual type using reflection and performing on the object some action (e.g., reading fields) that would be undetected by any user-level indirection mechanism. Next we discuss practical assumptions that let us classify different parts of system functionality for safe user-level indirection.

In Section 6.2 we distinguished several different kinds of events typically captured by user-level indirection: access to fields, method calls, constructor calls, and so forth. Clearly none of these events can be captured if they occur entirely within opaque code. For instance, it is impossible to capture updates to state (i.e., variables) that is defined inside native code. The interesting case, however, is that of events concerning user-level (i.e., non-opaque) entities and the question of whether these can occur inside opaque code. For instance, we may want to capture all updates to an object field that is declared in a Java system class implemented in bytecode. We need to ask if this field is ever accessed inside native code. In this section we assume the full gamut of user-level indirection events, including access and modification of fields. If a certain application is only interested in capturing method and constructor calls, the restrictions are typically far less severe. Nevertheless, most interesting applications of user-level indirection (esp. distributed execution and persistence) need to capture field accesses.

Here we abstract away the specifics of the J-Orchestra approach. It makes two main heuristic assumptions regarding system classes:

- Classes without native methods have no special semantics.
- Native methods do not use dynamic type discovery (reflection, downcasting, or any low-level type information recovery) on objects supplied through method arguments.

These assumptions generally hold true with few exceptions. The first assumption does not hold, for instance, for classes in the `java.lang.ref` package. The second assumption does not hold in the implementation of reflection classes themselves. In Section 6.5 we discuss a study of the Sun implementation of Java system classes and how it supports our assumptions.

The first assumption essentially states that the JVM is not allowed to handle different types of objects specially when the objects just use plain bytecode instructions. For instance, the JVM is not allowed to detect the construction of an object of a "special" type and keep a reference to this object that native code can later use for destructive state updates. This is a reasonable assumption, conforming to good software design practices. The second assumption states that native code is strongly typed: if a reference is declared to be of type `T`, it can never be used to access fields (method calls are fine) of a subclass of `T`. For instance, the assumption prohibits native methods from taking an `Object`-typed argument, checking if it is actually of a more specific type (e.g., `Thread` or `Window`), casting the object to that type and directly accessing fields or methods defined by the more specific type. This assumption also encodes a good design practice: code exploits the static type system as much as possible for correctness checking.

With the above two assumptions, we can perform a classification of Java system classes with respect to whether they can employ user-level indirection transparently or not, based on their usage of native code. We will use the term *NUI* (for *non-user-indirectible*) to describe classes that cannot employ user-level indirection transparently. We can generalize the J-Orchestra rules from Chapter IV to infer all classes that have user-level indirection limitations, as follows:

1) A system class with native methods is NUI.

2) A system class used as a parameter or return type for a method or static method in a NUI class is NUI.

3) If a system class is NUI, then all class types of its fields or static fields are NUI.

4) If a system class, other than `java.lang.Object`*, is NUI, then its subclasses and superclasses are NUI.*

(Just like in Chapter IV, the above rules represent the essence of the analysis but are not complete. For instance, they do not discuss arrays or exceptions—these are handled similarly to regular classes holding references to the array element type and method return types, respectively. Note that interface access does not impose restrictions since an interface cannot be used to directly access state. We prefer the abbreviated form of the rules for readability, especially since the analysis is based on heuristic assumptions, and therefore we do not make an argument of strict correctness.[2] The numbers we later report are for the full version of the rules, however.)

Rule 1 above is justified because no user-indirection technique can guarantee to capture all field updates of an instance of a class with a native method. The native method can always perform updates without any indirection.

2. In addition, one possible native dependency that might not be determined correctly through our type base heuristic is "intrinsics," a class loading mode in which a JVM ignores the bytecode file of a class defined in the platform specification, providing instead a native implementation for the class's functionality. The problem arises when the vendor of a JVM that uses "intrinsics" fails to mark intrinsic methods as native. In that case, the only way to find out if some methods are intrinsic is by trial-and-error, and our static type based heuristic is no longer sufficient. Empirically, this has not been an issue for Sun's Hot Spot JVM.

Rule 2 is justified with a similar argument: if an object can be passed to native code, native code can alias it and (either during the native method execution or during a later invocation) change its state. Furthermore, the rule can be applied transitively: if a class is NUI then we cannot replace all its uses with uses of an instrumented version in a user package `UP`. Then all objects used as arguments of any method (even non-native) may have their fields accessed directly.

Rule 3 is analogous to Rule 2 but for fields: native code can access any object transitively reachable from an object that leaks to native code.

Rule 4 is justified by the specifics of the J-Orchestra user-level indirection scheme. We saw an instance of this restriction in Section 6.3: if a class cannot be indirected, its uses in the application cannot instead employ a modified copy of the class in a user-level package. Thus, all subclasses and superclasses also cannot be copied to a user level package, as they may need to access non-public fields of their superclass.

These rules enable user-level indirection to be used safely for many Java system classes. Specifically, 37% of the Java system classes are classified as having no dependencies to native code and, thus, being able to employ user-level indirection safely.

Still, however, these rules are too conservative, as 63% of the system classes are deemed non-indirectible. Nevertheless, the rules are a good starting point and can be weakened to be made practical for specific applications of user-level indirection. For instance, in the context of J-Orchestra one more assumption is made relating to the way native code in different libraries can share state. The extra assumption allows placing different pieces of native code on separate machines and placing the instances of opaque classes in the same machine as the relevant code [51][87].

Next, we show one important general-purpose weakening of the rules. Rules 2 and 4 can be weakened significantly if we are allowed to modify system packages (still without touching native code) and we employ a more sophisticated user-level indirection scheme than that of J-Orchestra or TCH.

6.4.2 More Sophisticated Type-Based Analysis

The rules of the previous section are conservative because they assume that all code in system packages (be it native or not) is opaque. See, for instance, Rule 2: although any object that is used as a parameter of a native method can have its fields accessed with no indirection, there is no need to recursively propagate this constraint to the non-native methods of this object as well. If the object class is in pure bytecode, we can edit it and introduce indirection for accesses to its parameters. This, however, relies on a low-level assumption: we assume that the user-level indirection technique can modify system packages in order to edit the bytecode of existing system classes or add a new class in a system package. This is not desirable in some user-level indirection settings because it requires control over the startup environment of the JVM. Such control is not always possible, e.g., for deploying applets that random users will download and use inside a browser, or in systems in which the user cannot modify or extend the system package for security. Nevertheless, many applications of user-level indirection are allowed to set the parameters of the runtime system, and this can include a modified system package.

Under this assumption, we can use a weaker version of Rules 2 and 4.

1) A system class with native methods is NUI.

2') A system class used as a parameter or return type for a native method is NUI.

3) If a system class is NUI, then all class types of its fields or static fields are NUI.

4') If a system class is NUI, then its superclasses are NUI.

The weaker rules push the limits of user-level indirection much further: fewer than 8% of the Java system classes are classified as unable to employ user-level indirection (i.e., NUI). This means that a general-purpose user-level indirection technique can apply to more than 92% of the Java system classes with no special handling.

We already discussed how the new version of Rule 2 is a result of instrumenting the bytecode of bytecode-only NUI classes. The weakening of Rule 4 is more interesting. In the new Rule 4, a class does not impose any restrictions on its subclasses. This also eliminates any special handling of the `java.lang.Object` class, which is a common singularity in user-level indirection schemes.

To use the weaker version of Rule 4, we need to make sure that every system class `C` that cannot employ user-level indirection transparently is replicated in a user-level package. The replica class will just delegate all method calls to the original. Subclasses of `C` that have no native dependencies will employ full user-level indirection: an instrumented copy will be created in a user package and all references to the original class will become references to the instrumented version. As discussed in Section 6.3, the problem is that the instrumented class will not be able to access non-public members of `C`, as it is not in the same package as `C`. One solution is to make public all non-public members of class `C` by editing the class bytecode. (Or, equivalently, to create a subclass of `C` that exports the non-public members of `C`—see later.) A safer approach would be to emulate the Java access control at run-time using a technique such as that proposed by Bhowmik and Pugh [1] for

the Java inner classes rewrite. At load time, class `C` creates a secret key and passes it to the instrumented version of its subclass. When objects of the instrumented class need to access `C` members, they call a public method that also receives and checks the secret key. This is a safe emulation of the Java access protection, yet it avoids the requirement of placing classes in the same package.

An example application of this technique is shown in Figure 6-2(a). The example class `File` of Section 6.3 is now shown with a non-public field `field1`. `File` has a subclass `TXFile` with no native dependencies. Figure 6-2(b) shows the transformed classes so that `UP.File` and `UP.TXFile` can correctly replace all uses of `File` and `TXFile`, respectively, yet `UP.TXFile` can employ fully transparent user-level indirection. (As a low-level note, this transformation means that the instrumented system package, `UP`, needs to be loaded by the bootstrap class loader, since there is a call to method `UP.File.setKey` inside the `File` system class. The easiest way to effect this is to put the `UP` package in the `rt.jar` file.)

The effects of the transformation on the example class hierarchy are shown pictorially in Figure 6-3.

6.5 Validating The Assumptions and Analysis

We validate the assumptions and analysis of the previous section in three ways: first we measure the impact of our type classification for real applications: can we indeed use user-level indirection, without any special-case handling, for a large number of the system classes used by realistic applications? Next we examine an actual native code implementa-

```
class File {
 SomeT field1;
 ...
 public native void write(byte b);
 public void newLine() {...}
}

class TXFile extends File {
 ...
 public void writeString(String s) {
... foo(field1) ... }
}
```

Figure 6-2: (a): Original system class File (with a native method) and subclass TXFile (without native dependencies).

```
class File {
 SomeT field1;
 // Allow free access to field1 only to
 // class UP.File (and children)
 private static final Object key_ = new Object();
 static { UP.File.setKey (key_); }
 public SomeT get_field1(Object key) {
   if (key != key_)
     throw new IllegalAccessException();
   return field1;
 }
 ...
 public native void write(byte b);
 public void newLine() {...}
}

// Just delegates to File. Only used for correct
// subtype hierarchy.
class UP.File {
 protected File origImpl_;
 protected static Object  key_;
 public static void setKey(Object key)
 { key_ = key; }
 ...
 // delegate to native method
 public void write(byte b) { origImpl_.write(b); }
 public void newLine() { origImpl_.newLine(); }
}

class UP.TXFile extends UP.File {
 ...
 // methods of this class can employ any
 // user-level indirection scheme
 public void writeString(String s) {
   ...foo(origImpl_.get_field1(key_))...
 }
}
```

Figure 6-2: (b): Result of the user-level indirection transformation, with safe access to non-public fields of class File.

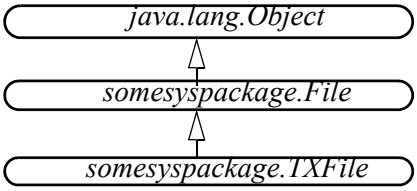


Figure 6-3: (a): A File class hierarchy

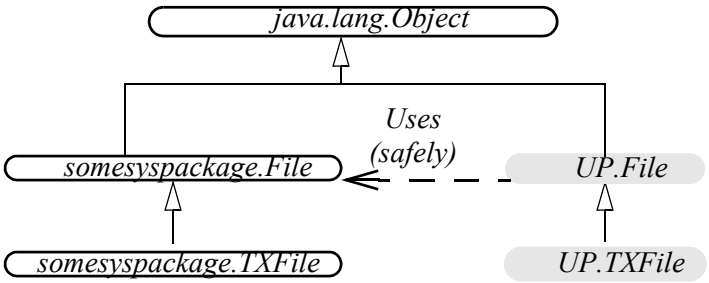


Figure 6-3: (b): Removing subclassing restrictions

tion of system methods and check whether it satisfies our assumptions. Finally, we perform a dynamic analysis of several Java applications and show that they do not violate the results of our type-based analysis during their execution.

6.5.1 Impact on Real Applications

An interesting question is to quantify the impact of the type-based analysis for real applications, as opposed to the set of all Java system classes. Although the more sophisticated version of our analysis allows to use indirection in 92% of the system classes, the remaining 8% are some of the most heavily used classes in practice. We demonstrate this in Table 6-1.. The table shows how many of the system classes actually used by different Java applications are classified as NUI under our analysis of Section 6.4.2. The table also shows how many of the used system classes have native methods themselves—this is a

lower bound on the number of NUI classes under any analysis. (We find the used classes by dynamically observing the loaded classes, minus JVM bootstrap classes. We then run our type-based analysis with the set of used classes as a universe set—any NUI dependencies introduced by classes that were not loaded are ignored.)

Three of the applications (javac, jess, mpegaudio) are standard benchmarks from SPEC JVM'98. (The rest of the SPEC JVM'98 programs yield practically identical numbers.) Unsurprisingly, these benchmarks are old and exercise few of the Java system classes. Nevertheless, we still see that more than 62% of the system classes used can employ user-level indirection. The next seven applications (antlr, bloat, chart, hsqldb, jython, ps, xalan) are from the more modern DaCapo benchmark suite (version beta050224). These applications are more realistic, yet they still do not exercise a large part of the Java system libraries. We see that our analysis enables 66-85% of the system classes used in the DaCapo benchmark programs to be safely indirected. For applications that exercise more of the Java system classes, we examined the Sun demo application SwingSet2 and the JBits FPGA simulator by Xilinx. The inputs used for these two applications were interactive and consisted of navigating extensively through the application's GUI and performing standard program actions (e.g., loading a simulator and an FPGA configuration and performing simulation steps). Both of these applications exercise over 1400 Java system classes. Only 21 and 16% (for JBits and SwingSet2, respectively) of these classes were found to be NUI under our analysis: the rest can employ user-level indirection without any special treatment. Finally, we include in our suite the RMIServer sample application from Sun, in order to exercise networking system classes.

Thus, Table 6-1. confirms that native code is not a negligible part of real applications. Additionally, although the type analysis assumes the most general native code behavior that respects its assumptions, it is still sufficient for enabling safe indirection for the large majority of Java system classes used in actual applications. (Where safety is always contingent on non-violation of our heuristic assumptions by the native code. We later discuss how we confirm that our approach is indeed safe for these executions.)

Table 6-1. Type-based analysis of used system classes

Application	#classes	#native	%native	#NUI	%NUI
javac	167	21	13	62	37
jess	165	21	13	61	37
Mpeg audio	158	21	13	60	38
Antlr	209	21	10	67	32
Bloat	275	25	9	80	29
Chart	601	69	11	194	32
Hsqldb	295	26	9	83	28
Jython	263	20	8	76	29
Ps	175	18	10	60	34
Xalan	505	21	4	74	15
SwingSet2	1887	120	6	303	16
JBits	1442	124	9	306	21
RMI Server	415	37	9	109	26

6.5.2 Accuracy of Type Information

Recall that one of the heuristic assumptions of our type-based analysis is that the APIs to system functionality offer accurate type information. That is, we assume that native code does not discover type information dynamically: if a native method signature refers to type `A`, then it does not attempt to dynamically discover which particular subtype of `A` is the actual type of the object and to use fields or methods specific to that subtype. It is certainly common to pass instances of subtypes of `A` to the native method, but these should only be

accessed using the general interface defined by the supertype `A`. This assumption is in line with good object-oriented design.

Although the assumption is soundly motivated, there are certainly exceptions in real code. Nevertheless, such exceptions are fairly rare. To validate the assumption, we examined part of the implementation of native methods in Sun's JDK 1.4.2. We searched for the use of specific idioms throughout native method implementations and we examined in detail all native methods (109 of them) accepting as argument or returning as result an object with declared type `java.lang.Object` (the root of the Java inheritance hierarchy). In our study, we observed few violations of our assumptions. The most important ones are:

- reflection functionality routinely circumvents the type system, as expected. Reflection requires special handling in a user-level indirection environment.
- passing primitive arrays to native code is typically invisible to the type system. Several native methods accept an `Object` reference but implicitly assume that they are really passed a Java array of bytes or integers. This does not affect our analysis, as we consider primitive types and their arrays to be non-indirectible by default.
- a handful of methods have poor type information and violate our type accuracy assumptions. For instance, method `socketGetOption` in class `java.net.PlainSocketImpl` takes an `Object` as argument, casts it into a `java.net.InetAddress` and then sets one of its fields. (The `addr` field is set when the method returns the bind address for its socket implementation.) Similarly, native method `getPrivateKey` in class `sun.awt.SunToolkit` assumes that its `Object` argument is really a `java.awt.Component` or a `java.awt.MenuComponent` and dynamically discovers its actual type.

  These exceptions, however, are very rare, in our experience. A quick search of all native code in Java system libraries (for all platforms together) reveals just 69 uses of the JNI function `IsInstanceOf`, which is the main way to do dynamic type discovery in native

code. In contrast, there are about 5900 uses of the Java counterpart, `instanceof`, in plain Java code in the system libraries. (The total size of Java code in system libraries is roughly twice the size of C/C++ native code, so the discrepancy is not justified by the size alone.)

We, thus, feel that our heuristic assumption is well-justified. Even though the native implementation is free to circumvent the type system, we believe that in practice it is reasonable to assume that sufficient type information exists at the user/system boundary of languages like Java to allow a heuristic but fairly good type-based analysis. Clearly the analysis will not offer strict guarantees, but if it determines that a certain system class can employ user-level indirection, it is highly likely to be right. We quantify this likelihood for actual applications next.

6.5.3 Testing Correctness

Our type-based analysis attempts a heuristic solution to an unsolvable problem. Recall that if we treat native code as an adversary, there are no safe assumptions we can make, other than "all native code can directly access and modify all objects". This assumption invalidates every kind of user-level indirection. Nevertheless, in practice our heuristic, type-based approach works well. (Our experience with J-Orchestra was what first suggested to us that a type-based analysis is sufficient for ensuring safe indirection in practice.)

We dynamically analyzed the applications discussed above to confirm that the results of our type-based analysis are rarely, if ever, violated in practice. We instrumented a Java VM to observe all reads and writes to object fields performed inside native code. Then we checked whether fields of a class that we did not consider NUI are ever read or written inside native code. Of course, this experiment is just a test under specific inputs.

Our analysis results could still be violated by different program inputs. Nevertheless, given the amount and variety of tested code and inputs, we have high confidence in our observations.

Almost all applications listed in Table 6-1. exhibit accesses to Java object fields from inside native code. Some applications (especially the more graphics-intensive ones) have native code access the fields of objects of more than 50 different classes. Throughout all executions of the applications, we observed only two instances of access inside native code to objects of types that were not classified as NUI. Both cases represented native code implementation patterns in Sun's JDK 1.4.2 that violated our type-accuracy assumptions.

Specifically, the first case was that of method `populateGlyphVector` in class `sun.awt.font.NativeFontWrapper` (not a directly user-accessible class). The method accepts a `java.awt.font.GlyphVector` parameter but implicitly assumes that the true type of the parameter is `sun.awt.font.StandardGlyphVector` and proceeds to set specific fields of that class. This is a classic case where information is not present in the type signatures of native methods for no apparent good reason. (Upon further inspection, a couple of more methods in the same class also circumvent the type system for `GlyphVector` arguments.)

The second case was that of the constructor of class `sun.java2d.loops.MaskFill`. The constructor accepts a `java.awt.Composite` parameter but assumes its real type is `java.awt.AlphaComposite`. Although this is again a bad practice of obscuring information from the type system, at least in this case there is some code economy benefit from doing so: the constructor is only called in native code using dynamic method discov-

ery (i.e., reflection at the native level). Eliding the specific type information allows the constructor to be called by the same code as some other similar constructors.

In summary, our experience confirms that a type-based analysis is quite safe in practice. Although no guarantees can be offered (as the assumptions can be violated by the implementation of native methods) one can reasonably expect that the type analysis will be safe. In the absence of complete information on the behavior of native code, our analysis is a clear win. The alternatives are to either not support indirection for any system classes, or to leave the user with no assistance in determining the correctness of applying indirection.

6.6 Conclusions

In recent years, the high and growing popularity of high-level languages such as Java and C#, running on top of virtual machine-based runtime systems, has influenced the proliferation of user-level indirection techniques for achieving systems-level extensibility. The ability to transform a piece of software automatically and correctly by enhancing it with useful functionality such as logging, persistence, distribution, and others, relieves the programmer from the necessity of performing tedious and error-prone tasks by hand. However, the applicability of all such user-level indirection techniques is limited by the presence of native code. This chapter has studied ways that identify these limitations, in order to enable user-level indirection to be applicable as widely as possible. In the greater scheme, this chapter has generalized one of the technical contributions of this dissertation to the domain of user-indirection-based software systems, having made the following observations:

- Native code can invalidate any user-level indirection technique in the worst case. Although this is a standard observation for program analysis experts, it is a topic often completely ignored by implementors of user-level indirection mechanisms.
- A simple type-based analysis together with fairly general assumptions can help distinguish classes that are safely indirectible from those that are not. It is interesting that the type information at the user/system boundary would be sufficient for this purpose. It is the type system of modern OO languages such as Java that is directly responsible for enabling this analysis. In other words, the analysis would be impossible at the user/system boundary between C and Unix or Windows, in which most of the arguments to system library calls are unstructured pointers and byte buffers.
- These findings have the potential to be of value in the design of future runtime systems and environments, making the code running on top of them easier to indirect. Specifically, these findings pointed out the need for a runtime specification that would describe how system classes interact with their native platform-specific libraries. Having such a runtime specification, perhaps in the form of annotations of Java system classes, would make user-level indirection techniques safe from the possibility of being invalidated by native code.

CHAPTER VII

RELATED WORK

The objective of this chapter is to put the research described in this dissertation into perspective by showing how it relates to existing work. First, we discuss directly related work as pertaining to each of the three software tools explored by this dissertation. Then we identify how this work on separating distribution concerns utilizes approaches and techniques from different research areas. Finally, we outline how this work could benefit or influence various areas of research and practice.

7.1 Directly Related Work

Because this dissertation takes a three-pronged approach to separating distribution concerns, consisting of middleware with copy-restore semantics, a program generator for distribution, and an automatic partitioning system, we similarly discuss directly related work as pertaining to each of these software tools.

7.1.1 NRMI

7.1.1.1 Performance Improvement Work

Several efforts aimed at providing a more efficient implementation of the facilities offered by standard RMI [80]. Krishnaswamy et al. [45] achieve RMI performance improvements by replacing TCP with UDP and by utilizing object caching. Upon receiving a remote call, a remote object is transferred to and cached on the caller site. In order for the

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

Several systems improve the performance of RMI by using a more efficient serialization mechanism. KaRMI [65] 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 [53][54] takes an alternative approach by using native code compilation to support compile and run time generation of marshalling code. It is interesting to observe that most of the optimizations aimed at improving the performance of the standard RMI and call-by-copy can be successfully applied to NRMI and call-by-copy-restore. Furthermore, such optimizations would be even more beneficial to NRMI due to its heavier use of serialization and networking.

7.1.1.2 Usability Improvement Work

Thiruvathukal et al. [85] propose an alternative approach to implementing a remote procedure call mechanism call for Java based on reflection. The proposed approach employs the reflective capabilities of the Java languages to invoke methods remotely. This simplifies the programming model since a class does not have to be declared `Remote` for its instances to receive remote calls.

While CORBA does not currently support object serialization, the OMG has been reviewing the possibilities of making such support available in some future version of IIOP [62]. If object serialization becomes standardized, both call-by-copy and call-by-copy-restore can be implemented enabling [in] and [in out] parameters passing semantics for objects.

The systems research literature identifies Distributed Shared Memory (DSM) systems as a primary research direction aimed at making distributed computing easier. Section 7.1.3 discusses DSM systems in greater detail. However, in comparison with NRMI, DSM systems can be viewed as sophisticated implementations of call-by-reference semantics, to be contrasted with the naive "remote pointer" approach shown in Figure 2-3 on page 18. On the other hand, the focus of DSM systems is very different from that of middleware. DSMs are a great enabling technology for distributed computing as a means to achieve parallelism. Thus, they have concentrated on providing correct and efficient semantics for multi-threaded execution. To achieve performance, DSM systems create complex memory consistency models and require the programmer to implicitly specify the sharing properties of data. In contrast, NRMI attempts to support natural semantics to straightforward middleware, which is always under the control of the programmer.

NRMI (and other mainstream distribution middleware systems) do not try to support "distribution for parallelism" but instead facilitate distributed computing in the case in which an application's data and input are naturally far away from the computation that needs them.

A special kind of tools that attempt to bridge the gap between DSMs and middleware are automatic partitioning tools such as our own J-Orchestra. (We discuss other automatic

partitioning systems in Section 7.1.3.) Such tools split centralized programs into several distinct parts that can run on different network sites. Thus, automatic partitioning systems try to offer DSM-like behavior but with more ease of use and compatibility: Automatically partitioned applications run on existing infrastructure (e.g., DCOM or regular unmodified JVMs) but relieve the programmer from the burden of dealing with the idiosyncrasies of various middleware mechanisms.

The JavaParty system [31][65] works much like an automatic partitioning tool, but gives a little more programmatic control to the user. JavaParty is designed to ease distributed cluster programming in Java. It extends the Java language with the keyword `remote` to mark those classes that can be called remotely. The JavaParty compiler then generates the required RMI code to enable remote access. Compared to NRMI, JavaParty is much closer to a DSM system, as it incurs similar overheads and employs similar mechanisms for exploiting locality.

Doorastha [19] represents another piece of work on making distributed programming more natural. Doorastha allows the user to annotate a centralized program to turn it into a distributed application. Although Doorastha allows fine-grained control without needing to write complex serialization routines, the choice of remote calling semantics is limited to call-by-copy and call-by-reference implemented through RMI remote pointers or object mobility. Call-by-copy-restore can be introduced orthogonally in a framework like Doorastha. In practice, we expect that call-by-copy-restore will often be sufficient instead of the costlier, DSM-like call-by-reference semantics.

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

subject [96]). The main point of criticism has been that distributed systems fundamentally differ from centralized systems because of the possibility of partial failure, which needs to be handled differently for each application. The “network transparency” offered by NRMI does not violate this principle in any way. Identically to regular RMI, NRMI remote methods throw remote exceptions that the programmer is responsible for catching. Thus, programmers are always aware of the network's existence, but with NRMI they often do not need to program differently, except to concentrate on the important parts of distributed computing such as handling partial failure.

7.1.2 GOTECH

Directly related work for GOTECH includes other tools that help in adding distribution, but without taking away from the programmer the control and responsibility of the distribution process. Such tools are “distributed programming aids”: they help do the tedious tasks that the programmer would otherwise need to do manually and that would “pollute” the code describing the application logic. Nevertheless, the programmer is still responsible for ensuring that the tools do the right job for the application at hand.

Indirectly related work includes mostly application partitioning tools and Distributed Shared Memory systems. Such tools offer a higher-level interface. Their user does not necessarily program the distributed application, but rather offers hints to improve its performance. These tools have a higher correctness responsibility: they attempt to correctly distribute any application although they usually result in loss of efficiency and are applicable in fewer situations than the “distributed programming aids”.

Many domain-specific languages have been proposed to aid distributed programming, and some of them [40][52] were key examples in the early steps of Aspect Oriented Programming. Such domain-specific languages for distribution are described in detail later, in Section 7.1.3, but we can make general observations regarding the GOTECH framework's advantages:

- it is an easy to evolve tool, based on widely used aspect-oriented infrastructure (AspectJ and XDoclet). Inspecting and changing the functionality of our XDoclet templates is much easier than changing the code for any of the above domain-specific tools.
- it employs NRMI as a unique way to support a remote call semantics that is closer to local execution. NRMI is applicable to many common scenarios, eliminating the need for explicitly updating data when changes are introduced by remote calls.
- GOTECH targets EJBs as a distribution substrate. This is a more complex, industrial-strength technology than the middleware used by previous systems.

Both DSMs and partitioning systems operate at a much higher level than tools like GOTECH. They strive for correct distributed execution of all applications and give the programmer much less control over distribution choices. Therefore, similarly to very high-level languages, these tools are valuable for the cases for which they are applicable, but these cases are a small part of the general distributed computing landscape. In contrast, GOTECH assists the programer in generating tedious code that would otherwise be intertwined with the application logic.

Finally, other researchers have examined the suitability of aspect-oriented techniques for different domains. For example, Kienzle and Guerraoui [43] examined the suitability of aspect-oriented tools for separating transaction logic from application logic. Separating transaction processing from application logic is very hard, and possible only

under very strict assumptions about the application. These findings of Kienzle and Guerraoui are consistent with longtime observations of the database community.

7.1.3 J-Orchestra

Much research work is closely related to J-Orchestra, either in terms of goals or in terms of methodologies, and we discuss some of this work next.

Several recent systems other than J-Orchestra can also be classified as automatic partitioning tools. In the Java world, the closest approaches are the Addistant [84] and Pangaea [74][75] systems. The Coign system [33] has promoted the idea of automatic partitioning for applications based on COM components.

All three systems do not address the problem of distribution in the presence of unmodifiable code. Coign is the only one of these systems to have a claim at scalability, but the applications partitioned by Coign consist of independent components to begin with. Coign does not address the hard problems of application partitioning, which have to do with pointers and aliasing: components cannot share data through memory pointers. Such components are deemed non-distributable and are located on the same machine. Practical experience with Coign showed that this is a severe limitation for the only real-world application included in Coign's example set (the Microsoft PhotoDraw program). The overall Coign approach would not be feasible for applications written in a general-purpose language (like Java, C, C#, or C++) in which pointers are prevalent, unless these applications have been developed following a strict component-based implementation methodology.

The Pangaea system [74][75] has very similar goals to J-Orchestra. Pangaea, however, includes no support for making Java system classes remotely accessible. Thus, Pan-

gaea cannot be used for resource-driven distribution, as most real-world resources (e.g., sound, graphics, file system) are hidden behind system code. Pangaea utilizes interesting static analyses to aid partitioning tasks (e.g., object placement) but these analyses ignore unmodifable (system) code.

The JavaParty [31][66] system is closely related to J-Orchestra. The similarity is not so evident in the objectives, since JavaParty only aims to support manual partitioning and does not deal with system classes. However, the implementation techniques of JavaParty are very similar to the ones of J-Orchestra, especially for the newest versions of JavaParty [31]. For distributed synchronization, JavaParty relies on KaRMI, a drop-in replacement for RMI, that maintains correct multithreaded execution over the network efficiently. In contrast, J-Orchestra implements distributed synchronization on top of standard middle-ware.

J-Orchestra bears similarity with such diverse systems as DIAMONDS [16], FarGo [32] and AdJava [23]. DIAMONDS clusters are similar to J-Orchestra anchored and mobile groups. FarGo groups are similar to J-Orchestra anchored groups. Notably, however, FarGo has focused on grouping classes together and moving them as a group. In fact, groups of J-Orchestra objects that are all anchored by choice could well move, as long as all objects in the group move. We have not yet investigated such mobile groups, however.

The pioneering work at MCC in the early 90s identified classes as suitable entities for performing resource allocation in distributed systems. The experimental system described in reference [15] uses class profiling as a guide for assigning objects to the nodes of a distributed system. J-Orchestra has fully explored the idea of resource-based partition-

ing at the class or group of classes level of granularity, demonstrating the feasibility and scalability of the approach.

Automatic partitioning is essentially a distributed shared memory (DSM) technique. Nevertheless, automatic partitioning differs from traditional DSMs in several ways. First, automatic partitioning systems such as J-Orchestra do not change the runtime system, but only the application. Traditional DSM systems like Munin [14], Orca [5][6], and, in the Java world, cJVM [3][4], and Java/DSM [102] use a specialized run-time environment in order to detect access to remote data and ensure data consistency. Also, DSMs have usually focused on parallel applications and require programmer intervention to achieve high-performance. In contrast, automatic partitioning concentrates on resource-driven distribution, which introduces a new set of problems (e.g., the problem of distributing around unmodifable system code, as discussed earlier). Among distributed shared memory systems, the ones most closely resembling the J-Orchestra approach are object-based DSMs, like Orca [5][6].

Mobile object systems, like Emerald [11][39] have formed the inspiration for many of the J-Orchestra ideas on object mobility scenarios. The novelty of J-Orchestra is not in the object mobility ideas but in the rewrite that allows them to be applied to an oblivious centralized application.

Both the D [52] and the Doorastha [19] systems allow the user to easily annotate a centralized program to turn it into a distributed application. Although these systems are higher-level than explicit distributed programming, they are significantly lower-level than J-Orchestra. The entire burden is shifted to the programmer to specify which semantics is valid for a specific class (e.g., whether objects are mobile, whether they can be passed by-

copy, and so forth). Programming in this way requires complete understanding of the application behavior and can be error-prone: a slight error in an annotation may cause insidious inconsistency errors.

7.2 Related Research Areas

This dissertation explores new software tools for separating distribution concerns, placing the work on the intersection of software engineering, programming languages, and distributed systems. While acknowledging that centralized and distributed programming model are fundamentally different and should be treated as such (e.g., as stated in the well-known "Note" by Waldo et al. [96]), we, nevertheless, recognize that evolving a distributed program by introducing distribution to an existing centralized program has become a common programming task [44]. Thus, this research draws its motivation from the inherent difficulties of the programming task of transforming a centralized, monolithic program into a distributed program.

This research is related to software engineering. First, it identifies the limits of automating the process of introducing distribution to existing centralized programs. Second, it provides better software tools, thereby contributing to improving programmers's productivity. Finally, it explores new software engineering approaches for emerging domains such as ubiquitous computing.

This research is related to programming languages. This research takes advantage of several techniques and approaches that were first utilized to address various challenges in programming language implementation. Specifically, the NRMI algorithm for call-by-copy-restore is influenced by copying garbage collectors [100]. Several implementation

facets of the J-Orchestra system (e.g., maintaining the semantics of various language entities in a distributed environment, bridging the local/remote differences in parameters passing semantics, introducing indirection in the presence of unmodifiable code in the runtime system) have commonalities with issues in programming languages research. GOTECH uses code generation, which has branched away from programming languages into an independent research area [50].

This research is related to distributed systems. The Remote Procedure Call (RPC) mechanism [10] remains a popular programming model for building distributed systems, even in the research domain. For example, van Nieuwpoort at al. [59] demonstrate the feasibility of using RMI for building parallel grid applications. This dissertation makes several improvements to the RPC mechanism. Specifically, NRMI is the first RPC system that offers a fully-general call-by-copy-restore semantics for linked data structures, and J-Orchestra introduces a novel technique for maintaining concurrency and synchronization constructs over RPC efficiently. In a broader sense, because this research focuses on synchronous, RPC-enabled remote communication, it is not immediately obvious how this work could advance some of the current state-of-the-art of distributed systems such as peer-to-peer systems, sensor networks, grid computing, autonomic computing, and sophisticated fault-tolerance mechanisms. Nevertheless, the software tools explored by this dissertation can be adapted to work and be beneficial for many experimental distributed systems as well. In other words, by further investing into this work on improving software tools for challenging domains, we can get closer to fulfilling the ambitious objective of addressing the disparity between the advances in high-performance system design and the practices in industrial software development.

7.3 Beneficiaries of This Research

The results of this research can benefit several areas of research and practice. Despite the fact that the primary contributions of this research are in the domain of software technologies for distributed computing, some of the ideas explored by this dissertation are applicable to other domains. We next describe in turn how this research contributes to the areas software engineering, programming languages, and distributed systems.

This research contributes to software engineering by having investigated novel software tools that facilitate challenging programming tasks. This work has demonstrated how a combination of advanced development techniques (such as code generation, code transformation, and bytecode engineering) can assist the programmer in developing complex computer systems in the field of distributed computing. At the same time, this research has developed techniques that can be of value in multiple domains. For example, Bialek at al. [9] have adapted some of the J-Orchestra techniques to partition programs so that they could support dynamic updates. As a sign of its practical value and broader impact, this research has influenced the designers of JBoss [68], the most popular open source application server. According to Marc Fleury, JBoss founder, the bytecode engineering techniques of J-Orchestra have been the inspiration for the Aspect-Oriented Programming features of JBoss 4. JBoss 4 employs bytecode engineering to add various non-functional pieces of functionality to POJOs (Plain Old Java Objects), similarly to J-Orchestra adding distribution to unaware centralized programs. Another potential beneficiary of this research is the domain of ubiquitous computing, which has been recognized as needing better software support [1]. This research has identified automatic partitioning as a promising approach to

prototyping ubiquitous computing applications and also pointed out how, in this domain, semi-automatic tools could be beneficial at the development stages beyond prototyping. The GOTECH framework has introduced the approach of combining generative and aspect-oriented techniques to alleviate tedious and error-prone programming tasks, which can be of software engineering value in domains other than distributed computing. As an example, the domain-independent MAJ tool [107] has followed and improved on the GOTECH approach. In addition, because of its ease of use in the presence of complex J2EE conventions, GOTECH has been suggested as a tool for software engineering education. While discussing approaches to teaching concepts of software adaptation in distributed object computing, Gray [26] identifies the GOTECH framework as a possible tool for exposing students to applications of aspect-oriented techniques.

This research contributes to programming languages. The insights of generalizing the J-Orchestra indirection machinery can be applied to designing the runtime libraries of future virtual machines. This research has identified that the presence of unmodifiable code in the runtime system of a bytecode application can significantly hinder what kind of indirection can be safely applied to that application. If one had information on how exactly native code libraries interact among them and with the bytecode of system classes, this would allow more flexibility in designing the user-level indirection machinery. Unfortunately, the straightforward way to get such information is to analyze the source code of the runtime system. This is complicated at best and unrealistic at worst (source code may not be available). The natural avenue for extending program analysis when source code is not available is to employ programmer-supplied annotations. These annotations would form a language for communicating implementation insights. One can draw on the findings of this

research to create such an annotations scheme that would reflect how exactly system classes interact with native libraries.

This research contributes to distributed systems. We have already mentioned the contributions to the RPC mechanism, a common paradigm for building distributed systems. In addition, automatic application partitioning provides a software technology answer to several difficult systems problems. For example, state-of-the-art event-delivery systems such as JECho [105][106] derive performance benefits by collocating event processing functionality with event producers. Specifically, JECho introduces a novel software abstraction called eager handlers that send parts of event handling functionality from the consumer to the supplier sites of an event. This effectively partitions event handling into two parts, the first executed by the sender and the second by the receiver. Such partitioning of events processing functionality could limit bandwidth consumption or reduce the computational costs, depending on a given cost model. With automatic application partitioning we can achieve similar benefits for larger-scale applications by placing code near the resource it manages. For example, we can partition a centralized application for distributed execution in such a way that a particular systems resource such as graphics can be produced and filtered on the same network site, while only the filtered version will be transferred over the network to the site on which it will be displayed on a graphical screen. More sophisticated scenarios for collocating resources with the code managing them can be achieved through object mobility.

CHAPTER VIII

FUTURE WORK AND CONCLUSIONS

The algorithms, techniques, and tools for separating distribution concerns, explored by this dissertation, present ample possibilities for future work. Each of the developed software tools can be further enhanced in terms of both its capabilities and applicability. Furthermore, some of the general insights gained from this research can be applied to domains other than distributed computing. Some of the future research directions, resulting from this work, have already been explored both by us [92] and other researchers [107]. We next present some of the ideas for future work for NRMI, GOTECH, and J-Orchestra. After discussing the future work directions, we reiterate the merits of this dissertation and present our conclusions.

8.1 NRMI Future Work

NRMI, with its call-by-copy-restore semantic that makes remote calls look like local calls for stateless servers and single thread clients, is a convenient building block for other middleware facilities that emphasize ease of use without jeopardizing performance. Specifically, we would like to take our work on NRMI in the directions of greater generality and adaptability to network outages. The first direction will extend NRMI to explore a general problem of efficiently synchronizing a subset of the client state against a subset of the server state by means of a remote procedure call. This problem is common in enterprise

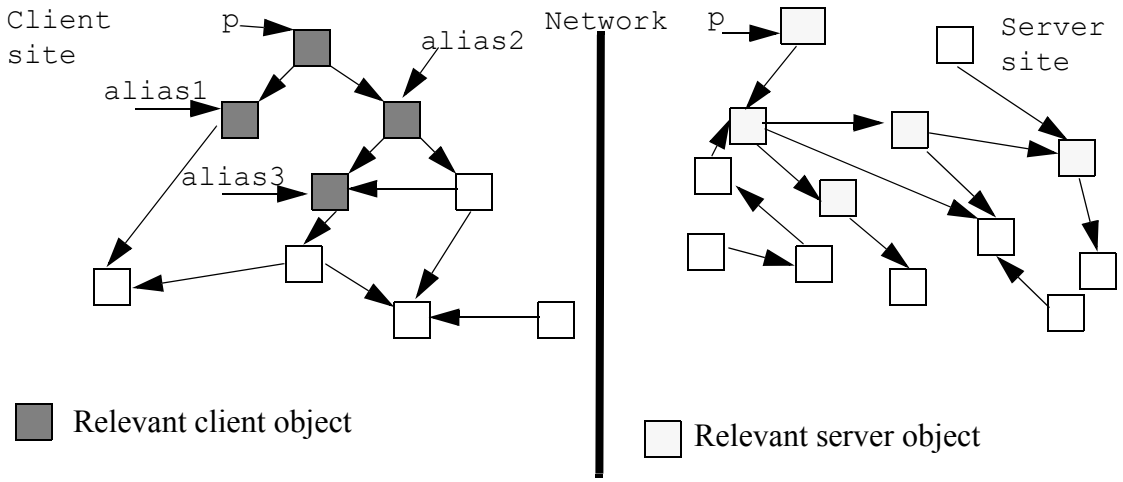


Figure 8-1: (a): A general remote call mechanism: a subset of the client heap, reachable from `p`, can be sent to the server, to be updated against a subset of the server heap.

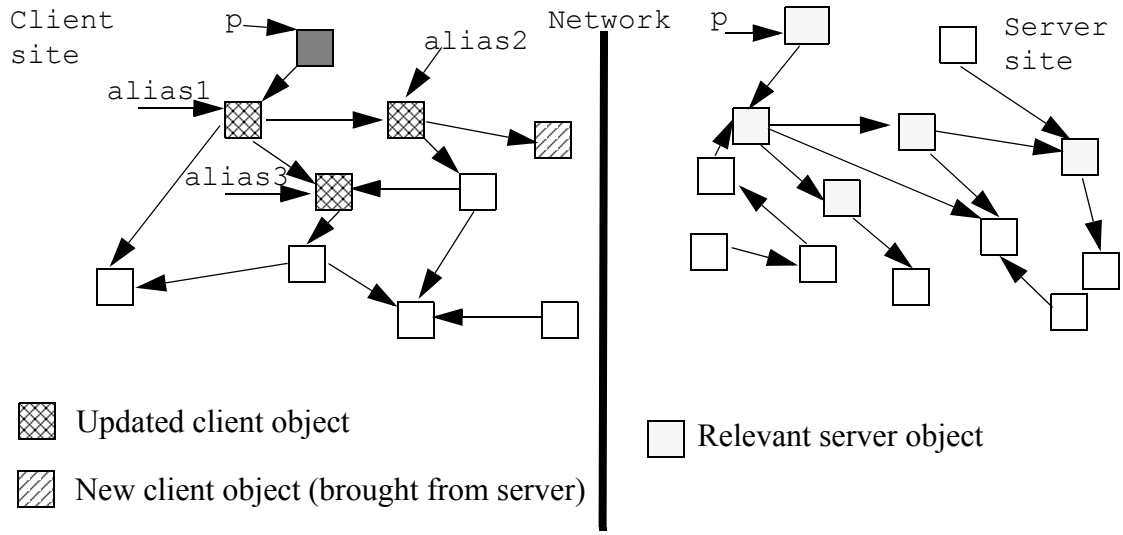


Figure 8-1: (b): A general remote call mechanism: param `p` is returned to the client and restored in place.

computing such as the domain captured by the J2EE specification [78]. Often a server environment contains a very large state of heap-allocated objects and clients need to synchronize themselves periodically against this state. Currently, no existing mainstream technology provides a convenient programming mechanism that implements this function-

ality. As a result, programmers resort to ad-hoc solutions that are error-prone and difficult to maintain and extend.

In abstract terms, the solution can be provided by an efficient implementation of a relaxed version of call-by-copy-restore semantics. For lack of a better term, we will call the mechanism that implements this semantic a "general remote procedure call." Figure 8-1 (a and b) demonstrates the desired behavior. One can provide an efficient implementation of a general remote procedure by reusing NRMI with its ability to update objects in place (i.e., preserving all the aliases) and extending it with a customizable serialization mechanism. NRMI already relies on Java Serialization [79], which provides the `transient` keyword to indicate a field that is not part of an object's persistent state and should not be serialized. Nevertheless, the `transient` keyword is too crude a mechanism for providing a truly customizable serialization as would be needed by the general remote procedure call mechanism. One interesting question that is to be explored is how this customizable serialization mechanism can be best expressed by the programmer. Perhaps it can be accomplished through special purpose annotations or even via the means of a domain specific language (DSL). Another question is how easy would that be to integrate this general remote procedure call mechanism into an application server environment such as the one provided by JBoss. Finally, it would be important to determine whether a general remote procedure call implementation can be optimized enough for real-world use.

Another future direction for NRMI work would be providing an adaptable middleware mechanism that could respond to network outages. This mechanism would have the potential to enhance data availability and the overall quality of service (QoS) in unreliable networks such as dynamic mobile wireless networks. Because of their ad-hoc nature such

networks are volatile and can temporarily become disconnected. Furthermore, usually outages in such networks are temporary and short in duration. If such a network outage happens during a remote call, the client computation might proceed up to the point when the data returned by the remote call is first referenced, which might be at some later point in the control flow than immediately following the remote call. This mechanism would enable continuing computation while the network is temporarily unavailable, and, in the presence of frequent but short interruptions, can result in improved throughput. Of course, this adaptable mechanism would be applicable to regular call-by-copy semantics in remote calls as well, taking into consideration only the return value of the call. However, with call-by-copy-restore, which also changes the values of the parameters of a remote call upon return, the problem becomes more comprehensive—solving it would require taking into consideration all the variables that could have been changed as a result of a remote call.

At the implementation level, realizing an adaptable middleware mechanism that could respond to network outages will require a combination of static analysis and code rewriting. Such static analysis techniques as control flow can determine conservatively the actual statements referencing by-copy-restore parameters and the return value of a remote call in the client portion a program that follows it. Then the code can be automatically rewritten to delay the blocking, occurring as a result of a temporary network outage during a remote call, up to the program statements determined through the analysis. Of course, extensive benchmarking at both the micro and macro levels would be required to determine how successful such a middleware mechanism can be in improving the throughput of applications operating in volatile network environments.

8.2 GOTECH Future Work

The GOTECH framework is one of the first research projects that has taken the approach of combining generative and aspect-oriented techniques. The GOTECH approach can be enhanced in several directions such as improving the framework and providing tools that would make it applicable in domains other than distribution. A representative of the latter direction is a recent work by my colleagues on a generator called Meta AspectJ (or MAJ for short) [107]. Their work has successfully resolved one of the major shortcomings of the GOTECH approach—its reliance on text-based templates.[1] As an evolutionary improvement of the GOTECH approach, MAJ represents the generated code as a typed data structure instead of arbitrary text, generating syntactically correct AspectJ programs. While the MAJ project has focused on providing a general-purpose generator, it would be interesting to explore how well a combination of generative and aspect-oriented techniques can help solve problems in domains other than distribution, with persistence and security being most promising.

Another future direction would be providing more mature support for the conversion of plain objects to EJBs with different tools. For instance, the JBoss AOP framework performs bytecode engineering at class load time to retrofit existing classes so that they become EJBs. This approach can be applied both to distribution and to persistence concerns and is of high industrial value. Since NRMI has already been implemented to work with JBoss, this bytecode engineering work can result in a replication of the GOTECH capabil-

1. Reliance on text-base templates is not a serious issue for GOTECH per se, which is a domain-specific generator with a fixed set of templates, but it definitely becomes so for any software generator that aims at generalizing the GOTECH approach.

ities at load-time. Finally, another promising direction for more mature use of GOTECH includes developing analysis tools that formalize the preconditions for the applicability of the approach and ensure they are met by a specific application.

8.3 J-Orchestra Future Work

J-Orchestra is the largest and most comprehensive software tool for separating distribution concerns explored by this dissertation both in terms of the actual distribution concerns that it successfully separates and in terms of the various case studies to which it has been applied. It is natural, therefore, that our work on J-Orchestra has led to multiple and diverse future work directions. Since it would be unrealistic to describe all of these future work directions in detail here, we outline some of the major ones next. These directions fall into two major categories: expanding the boundaries of application partitioning and applying the insights gained from the J-Orchestra project to domains other than distribution.

While J-Orchestra has demonstrated that automatic application partitioning is a viable technology for introducing distributed capabilities to a specific class of centralized applications, future work can address various limitations and shortcoming of the J-Orchestra approach. One inherent limitation of J-Orchestra has to do with the automatic nature of its approach. That is, the J-Orchestra user works at the class or group-of-classes level of abstraction. Thus, our approach is quite automatic and involves no programming, just resource-location assignment—for example, that graphics code should run on this machine, or the main engine should run on that machine. In contrast, a semiautomatic approach could let the user annotate detailed parts of the code and data, to indicate, for example, what data should be replicated, how the copies should remain consistent, and how leases should be

used for fault tolerance. Thus, a semiautomatic approach could resolve many of the issues associated with automatic partitioning.

One of such issues is that, in its present state, automatic partitioning does not offer any assistance in supporting highly dynamic interactions between communicating entities, which are common in ubicomp applications [97]. For example, ubicomp applications often allow for resources and services to come and go dynamically as users and devices enter and leave the environment. Because automatic partitioning does not change the original centralized application's logic or structure, flexibility and configurability must be designed into the original application before it is partitioned. In contrast, a semiautomatic approach could potentially support dynamic interactions through automatic modification of an unsuspecting application.

In general, a partitioning system tries to automate many hard distribution tasks. Any automation effort, however, hinders complete control for users with advanced requirements. Such requirements might include replication for fault tolerance; high performance through load balancing, caching, or asynchronous communication; security; and persistence. In an automatically partitioned application, it is not easy to use replication for redundancy and switch to a different server once a failure is detected. The conventional wisdom in the distributed-systems community is that mechanisms for handling distributed failure are extremely application-specific and can not be automated completely.

Again, the appropriate solution might be to follow a semiautomated approach, providing tool support for replication, load balancing, security, and so forth. In this way, the programmer would be relieved of the low-level complexity but would still be responsible for annotating parts of the code in detail and for the distribution's conceptual consistency.

In fact, Section 4.5.4 has described how J-Orchestra supports a semiautomated approach that enables the user to specify complex schemes for object mobility (e.g., "move this object whenever it is reachable from an argument of a remote method"). Nevertheless, because this is not a GUI-accessible feature, the user must write Java code that follows J-Orchestra framework conventions to enable such object mobility.

At the implementation level, a semiautomatic approach could, for example, let the user annotate the application code to express desired policies for data consistency in the context of possible failures. These annotations would form a domain-specific language for specifying properties of dynamic distribution. For instance, one could annotate a certain data field to indicate that many instances of it might exist. Another annotation could specify the leases that each client holds and the data that depend on each lease. The low-level code would then be generated from the annotations instead of having to be handwritten. Overall, the approach would be very similar to the one currently followed by the GOTECH framework, but it would also involve the J-Orchestra analysis and bytecode transformation engines, making it more powerful.

J-Orchestra currently uses a type-based analysis heuristic that determines which references can leak to which code. This heuristic is too conservative and its precision and sophistication can be improved. Specifically, one promising direction would be to expand it with various static analysis techniques such as control-flow and data-flow that would help determine with a greater degree of precision which references can leak to which native code. Of course, as we demonstrated in Chapter VI, any solution to this problem would be an approximation, and one has to make reasonable assumptions to account for both the inherent limitation of the existing static analyses techniques and the unpredictability of

native code behavior. Nevertheless, a more sophisticated analysis engine would enable more powerful rewrites.

One of such rewrites could support object-based partitioning, which would be orders of magnitude more fine-grained than the current class or group-of-classes abstraction level at which J-Orchestra operates. Of course a full object-based partitioning approach would not scale to realistic applications, but, when applied to only a limited subset of classes, it would be an extremely valuable addition to the existing J-Orchestra tool set. J-Orchestra already supports a limited version of object-based partitioning based on the objects' creation sites. Nevertheless, in this case, it is entirely up to the programer to ensure that such object-based partitioning makes sense. An object-based analysis could provide the programmer with information about how particular objects are used in the program, enabling more sophisticated partitioning scenarios.

Another tool that could empower the programer in making more informed partitioning decisions is the J-Orchestra profiler. In its current stage, the J-Orchestra profiler provides a very crude kind of information and as such offers several directions for future work. An important issue with profiling concerns the use of off-line vs. on-line profiling. Several systems with goals similar to ours (e.g., Coign [33] and AIDE [56]) use on-line profiling in order to dynamically discover properties of the application and possibly alter partitioning decisions on-the-fly. So far, we have not explored an on-line approach in J-Orchestra, because of its overheads for regular application execution. Since J-Orchestra has no control over the JVM, these overheads can be expected to be higher than in other systems that explicitly control the runtime environment. Without low-level control, it is hard to keep such overhead to a minimum. Sampling techniques can alleviate the overhead (at the

expense of some accuracy) but not eliminate it: some sampling logic has to be executed in each method call, for instance. Another issue has to do with fine-tuning the technique for analyzing the profiling results. This technique, given some initial locations and the profiling results, should determine a good placement for all classes. The technique that J-Orchestra currently follows is a clustering heuristic that implements a greedy strategy. It would be interesting to experiment with replacing this clustering heuristic with other algorithms that could provide a reasonable approximation, particularly for the situations when the number of partitions is greater than two.

In its current implementation, J-Orchestra treats security as an orthogonal concern. On the one hand, in designing the J-Orchestra rewrite engine, which transforms a centralized program into its distributed counterpart, we have made every effort not to introduce security vulnerabilities if at all possible. At the same time, we did not have an opportunity to have our rewrites follow a well-defined security policy. Producing a secure distributed system as the partitioning's end product has not yet been one of the primary objectives of J-Orchestra. Nevertheless, partitioning presents many interesting security challenges. Some prior work had focused on secure program partitioning [104], which is different from the problem of applying a security policy to resource-based partitioning. By splitting up the functionality of a centralized program to run on multiple network sites, talking to each other over the network, some information that would have never left the confines of a single address space, suddenly can get transferred over the network. Producing a coherent security policy and incorporating it into each and every step in the partitioning process could be an interesting research direction.

Aside from distribution, some of the insights, gained from our work on J-Orchestra, can be generalized to other domains. In abstract terms, the J-Orchestra approach can be described as adding capabilities to existing programs through bytecode modifications. In the case of J-Orchestra, the added capabilities are distribution. In the past, bytecode manipulations have been used to add other capabilities to existing programs, including persistence, profiling, logging, and so forth. Nevertheless, our work on J-Orchestra has achieved results that distinguish it from other work in its handling of the bytecode/native code interactions in the runtime system. Therefore, it would be beneficial to generalize our techniques from the domain of distribution to other domains, particularly the ones that have already employed bytecode transformations in the past. The indirection machinery of J-Orchestra can be generalized in a completely domain-independent way, resulting in a tool that would allow adding capabilities to existing programs by modifying the bytecode not only of application classes but also of system classes, whenever possible. One interesting application of this tool would be extending AspectJ with capabilities to apply aspects to systems classes.

In general, applying bytecode transformations can yield benefits in a variety of domains and software development scenarios. We have attempted to generalize the technique by exploring the idea of binary refactoring [92], which applies refactoring transformations (e.g., split class, glue classes, inline method, remove design pattern indirection) to a software application without affecting its source code. Binary refactoring is only the tip of the iceberg, but it demonstrates an important principle that a good program transformation approach should follow: program transformation should not sacrifice software maintainability in order to achieve performance or temporary convenience. It would be

interesting to see how program generation and transformation can be applied to large-scale program modifications.

8.4 Merits of the Dissertation

This dissertation has explored algorithms, techniques, and tools for separating distribution concerns. We discussed the motivation, design, and implementation of three software tools: NRMI, GOTECH, and J-Orchestra. We also identified the applicability issues of these tools and presented validation through case studies. We next reiterate some of the conceptual contributions of this dissertation.

1. *A general algorithm for call-by-copy-restore semantics in remote procedure calls for linked data structures*. The NRMI middleware mechanism provides a fully-general implementation of call-by-copy-restore semantics for arbitrary linked data structures, used as parameters in remote procedure calls.

2. *An analysis heuristic that determines which application objects get passed to which parts of native (i.e., platform-specific) code in the language runtime system for platform-independent binary code applications*. The J-Orchestra system utilizes this analysis heuristic to enable partitioning of unaware programs in the presence of unmodifiable native code in the runtime system. We also discuss how this heuristic can be fine-tuned and applied to other domains.

3. *A technique for injecting code in platform-independent binary code applications that will convert objects to the right representation so that they can be accessed correctly inside both application and native code*. The J-Orchestra system implements this technique in its rewrite for classes with native dependencies.

4. *An approach to maintaining the Java centralized concurrency and synchronization semantics over remote procedure calls efficiently.* The J-Orchestra system follows this

approach to transform centralized concurrency and synchronization Java constructs for distributed execution.

5. *An approach to enabling the execution of legacy Java code remotely from a web browser.* This approach is called appletizing, and it is fully realized as a specialization of automatic partitioning in the J-Orchestra system.

8.5 Conclusions

This dissertation has discussed research that is concerned with developing and evaluating software tools for separating distribution concerns. The goal of this research is to introduce software tools working with standard mainstream languages, systems software, and virtual machines that effectively and efficiently separate distribution concerns from application logic for object-oriented programs that use multiple distinct sets of resources. We believe that this research will contribute to the development of versatile tools and technology with practical value, innovative designs, and the potential to become mainstream in the future.

It is an exciting time to be a researcher in the field of software technology. For the first time in the history of computing, we have mainstream commercial languages such as Java and C# that are virtual machine based, platform-independent, garbage-collected, fairly type safe, conducive to good software engineering practices, and easily amenable to code transformation and generation. In addition, programs written in these languages show good and improving performance, thanks to the ever more sophisticated Just-in-Time compilation technologies. As a consequence, many interesting research developments in software technology, before applied to and tested on exclusively esoteric, research-only language environments, will be transferred to mainstream software development at ever accelerating

rates. All this makes software technologies a highly-dynamic research area with the potential of influencing how we build software today and in the future, and, hopefully, the contributions of this dissertation are a concrete step in realizing this vision.

REFERENCES

[1] Gregory D. Abowd, "Programming Environments … Literally: Ubicomp's Grand Challenge for Software Engineering," *ACM SIGSOFT*, ACM Press, 2002; `www.cra.org/Activities/grand.challenges/abowd.pdf`.

[2] Bowen Alpern, Anthony Cocchi, Stephen Fink, David Grove, and Derek Lieber, "Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[3] Yariv Aridor, Michael Factor, and Avi Teperman, "CJVM: a Single System Image of a JVM on a Cluster," in *International Conference on Parallel Processing (ICPP),* 1999.

[4] Yaviv Aridor, Michael Factor, Avi Teperman, Tamar Eilam, and Assaf Schuster, "A high Performance Cluster JVM Presenting a Pure Single System Image," In *ACM JavaGrande 2000*, 2000.

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

[6] Henri E. Bal and M. Frans Kaashoek, "Object Distribution in Orca Using Compile-Time and Run-Time Techniques," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1993.

[7] John K. Bennett, "The Design and Implementation of Distributed Smalltalk," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1987, volume 22 of SIGPLAN Notices, pages 318-330, 1987. Also available as Technical Report 87-04-02, University of Washington, Seattle, April 1987.

[8] Anasua Bhowmik and William Pugh, "A Secure Implementation of Java Inner Classes," *PLDI 99 poster session*.

[9] Robert Bialek, Eric Jul, Jean-Guy Schneider, and Yan Jin, "Partitioning of Java Applications to Support Dynamic Updates," *11th Asia-Pacific Software Engineering Conference (APSEC),* 2004.

[10] Andrew D. Birrell and Bruce Jay Nelson, “Implementing Remote Procedure Calls,” *ACM Trans. CS*, 2(1):39{59}, February 1984.

[11] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, “Distribution and Abstract Types in Emerald,” in *IEEE Trans. Softw. Eng.*, 13(1):65-76, 1987.

[12] Kumar Brahnmath, Nathaniel Nystrom, Antony Hosking and Quintin Cutts, “Swizzle Barrier Optimizations for Orthogonal Persistence in Java,” proc. *8th International Workshop on Persistent Object Systems (POS8) and 3rd International Workshop on Persistence and Java (PJW3)*, 1998.

[13] Jon Byous, “Opportunities Everywhere,” See `http://java.sun.com/javaone/general_sessions1.html`.

[14] John B. Carter, John K. Bennett, and Willy Zwaenepoel, “Implementation and performance of Munin,” in *13th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 152-164, October 1991.

[15] Arun Chatterjee, "The Class is an Abstract Behaviour Type for Resource Allocation of Distributed Object-Oriented Programs"; unpublished paper presented at the *OLDA-2 workshop at OOPSLA-92* (briefly described in the workshop report "Objects in Large Distributed Applications II" by Peter Dickman, ACM SIGPLAN OOPS Messenger, Vol 4 #2, (Addendum to the proceedings of *OOPSLA 1992*), pp 63--69, ACM Press).

[16] S.E. Chidamber and C.F. Kemerer, “A MetricsSuite for Object-Oriented Design,” IEEE Transactions on Software Engineering, 20(6):476–493, 1994.

[17] Gary Craig, Umesh Bellur, Kevin Shank, and Doug Lea, “Clusters: A Pragmatic Approach Towards Supporting a Fine Grained Active Object Model in Distributed Systems,” in *the 9th International Conference on Systems Engineering*, Las Vegas, 1993.

[18] Markus Dahm, “Byte Code Engineering," *JIT* 1999.

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

[20] Edsger W. Dijkstra, “On the role of scientific thought,” EWD 447, August 1974. Also in *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.

[21] Michael Factor, Assaf Schuster and Konstantin Shagin, “JavaSplit: A Runtime for Execution of Monolithic Java Programs on Heterogeneous Collections of Commodity Workstations,” *2003 International Conference on Cluster Computing (CLUSTER),* 2003.

[22] Michael Factor, Assaf Schuster and Konstantin Shagin, “Instrumentation of Standard Libraries in Object-Oriented Languages: the Twin Class Hierarchy Approach,” *Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2004.

[23] Mohammad M. Fuad and Michael J. Oudshoorn, “AdJava— Automatic Distribution of Java Applications,” in *the 25th Australasian Computer Science Conference (ACSC)*, 2002.

[24] Michael Garey and David Johnson, *Computers and Intractability—A Guide to the Theory of NP-Completeness*, W.H. Freeman, New York, 1979.

[25] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification*, Second Edition, Addison Wesley, 2000.

[26] Jeff Gray, “A Java-based Approach for Teaching Principles of Adaptive and Evolvable Software,” *Science of Computer Programming,* special issue on Practice and Experience with Java in Education (Qusay Mahmoud, ed.)*,* 2004.

[27] Steven A. Guccione, Delon Levi and Prasanna Sundararajan, “JBits: A Java-based Interface for Reconfigurable Computing,” in *the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999. See also `http://www.xilinx.com/products/jbits/` .

[28] M.H. Halstead, *Elements of Software Science*, Elsevier, 1977.

[29] W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1993.

[30] Bernhard Haumacher, Thomas Moschny, Jürgen Reuter, and Walter F. Tichy, "Transparent Distributed Threads for Java," in *the 5th International Workshop on Java for Parallel and Distributed Computing in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS 2003),* Nice, France, April, 2003.

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

[32] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit, "Dynamic Layout of Distributed Applications in FarGo," in *Int. Conf. on Softw. Engineering (ICSE)* 1999.

[33] Jarle Hulaas and Walter Binder, "Program Transformations for Portable CPU Accounting and Control in Java," *Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 2004.

[34] Galen C. Hunt, and Michael L. Scott, "The Coign Automatic Distributed Partitioning System," *3rd Symposium on Operating System Design and Implementation (OSDI'99)*, pp. 187-200, New Orleans, 1999.

[35] Jarminator: Free software application. From `http://www.javasvet.net/prj/jarminator/`

[36] JNotepad: Free software application. From `http://www.pscode.com/`

[37] JSR 220: Enterprise JavaBeans™ 3.0, `http://jcp.org/en/jsr/detail?id=220.`

[38] JStyle 5: Automated Java Source Code Metrics Analysis Application, Codework Inc., `http://www.codework.com/JStyle/product.html.`

[39] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black, “Fine-Grained Mobility in the Emerald System,” *ACM Trans. on Computer Systems*, 6(1):109-133, February 1988.

[40] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, “Aspect-Oriented Programming,” in *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[41] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, “An Overview of AspectJ,” in *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[42] Thilo Kielmann, Philip Hatcher, Luc Boug¹e, and Henri E. Bal, “Enabling Java for High Performance Computing: Exploiting Distributed Shared Memory and Remote Method Invocation,” *Communications of the ACM*, 44(10):110-117, 2001.

[43] Joerg Kienzle and Rachid Guerraoui, “AOP: Does It Make Sense? The Case of Concurrency and Failures,” in *European Conference on Object-Oriented Programming (ECOOP)*, 2002.

[44] Nelson King, “Partitioning Applications,” DBMS and Internet Systems Magazine, May 1997. See `http://www.dbmsmag.com/9705d13.html`.

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

[46] Gordon Landis, Charles Lamb, Tim Blackman, Sam Haradhvala, Mark Noyes, and Dan Weinreb, “ObjectStore/PSE: a Persistent Storage Engine for Java,” proc. *2nd International Workshop on Persistence and Java (PJW2)*, p. 129-137, 1997.

[47] Doug Lea, “Concurrent Programming in Java -- Design Principles and Patterns,” Addison-Wesley, Reading, Mass., 1996.

[48] Han B. Lee and Benjamin G. Zorn, “Bytecode Instrumentation as an Aid in Understanding the Behavior of Java Persistent Stores,” *OOPSLA 1997 Workshop on Garbage Collection and Memory Management*.

[49] Tobin J. Lehman, Alex Cozzi, Yuhong Xiong, Jonathan Gottschalk, Venu Vasudevan, Sean Landis, Pace Davis, Bruce Khavar, and Paul Bowman, “Hitting the Distributed Computing Sweet Spot with TSpaces,” *Computer Networks*, 35(4): 457–472, 2001.

[50] C. Lengauer, D. Batory, C. Consel,and M. Odersky (eds.), D*omain-Specific Program Generation*, Lecture Notes in Computer Science(LNCS) 3016, Springer-Verlag, 2004.

[51] Nikitas Liogkas, Blair MacIntyre, Elizabeth D. Mynatt, Yannis Smaragdakis, Eli Tilevich, and Stephen Voida, "Automatic Partitioning: Prototyping Ubiquitous-Computing Applications," *IEEE Pervasive Computing*, 3(3):40-47, July-September 2004.

[52] Cristina Videira Lopes and Gregor Kiczales, “D: A Language Framework for Distributed Programming,” PARC Technical report, February 97, SPL97-010 P9710047.

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

[54] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger Hofman, “Efficient Java RMI for Parallel Programming,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747-775, November 2001.

[55] Blair MacIntyre, Elizabeth Mynatt, Stephen Voida, Klaus Hansen, Joe Tullio, and Gregory Corso, “Support for multitasking and background awareness using interactive peripheral displays,” in *ACM Symposium on User Interface Software and Technology (UIST)*, 2001.

[56] Alan Messer, Ira Greenberg, Philippe Bernadat, Dejan Milojicic, Deqing Chen, T.J. Giuli, Xiaohui Gu, “Towards a Distributed Platform for Resource-Constrained

Devices," in *International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[57] Nataraj Nagaratnam, Arvind Srinivasan, and Doug Lea, "Remote objects in Java," in *Proceedings of IASTED '96, International Conference on Networks,* 1996.

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

[59] Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema, "Wide-Area Parallel Programming using the RemoteMethod Invocation Model," *Concurrency: Practice and Experience*, 12(8):643–666, 2000.

[60] ObjectDesign Inc., *ObjectStore PSE/PSE Pro for Java API User Guide*, 1999.

[61] Object Management Group, *The Common Object Request Broker: Architecture and Specification,* Revision 2.5, September 2001.

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

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

[64] Harold Ossher and Peri L. Tarr, "Using Multidimensional Separation of Concerns to (re)shape Evolving Software," *Communications of the ACM*, 44(10):43-50, 2001.

[65] Michael Philippsen, Bernhard Haumacher, and Christian Nester, "More Efficient Serialization and RMI for Java," *Concurrency: Practice & Experience*, 12(7):495-518, May 2000.

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

[67] Ingo Rammer, "Advanced .NET Remoting," *APress*, 2002.

[68] Francisco Reverbel and Marc Fleury, “The JBoss Extensible Server,” in *ACM Middleware 2003 Conference,* 2003.

[69] Tristan Richardson, Quentin Stafford-Fraser,Kenneth R. Wood and Andy Hopper, "Virtual Network Computing," *IEEE Internet Computing*, 2(1):33-38, 1998.

[70] Robert W. Scheifler, and Jim Gettys, “The X Window System,” *ACM Transactions on Graphics*, 5(2): 79-109, April 1986.

[71] Robert W. Scheifler, “X Window System Protocol, Version 11,” *Network Working Group RFC 1013*, April 1987.

[72] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented SoftwareArchitecture: Patterns for Concurrent and Networked Objects,* Wiley, 2000.

[73] Sergio Soares, Eduardo Laureano, Paulo Borba, “Implementing Distribution and Persistence Aspects with AspectJ,” in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002.

[74] Andre Spiegel, *Automatic Distribution of Object-Oriented Programs*, Ph.D. Thesis. FU Berlin, FB Mathematik und Informatik, December 2002.

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

[76] Andre Spiegel, “Pangaea: An Automatic Distribution Front-End for Java”, in *4th IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99)*, 1999.

[77] Sun Microsystems. Enterprise JavaBeans (EJB) Specification. Version 2.0. 2001. `http://java.sun.com/products/ejb/.`

[78] Sun Microsystems. Java 2 Enterprise Edition. `http://java.sun.com/j2ee/.`

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

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

[81] Sun Microsystems, The Java Tutorial, "How to Use Threads," `http://java.sun.com/docs/books/tutorial/uiswing/misc/threads.html#SwingWorker`.

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

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

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

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

[86] Eli Tilevich and Yannis Smaragdakis, "Automatic Application Partitioning: The J-Orchestra Approach," in *8th ECOOP Workshop on Mobile Object Systems*, 2002.

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

[88] Eli Tilevich and Yannis Smaragdakis, "NRMI: Natural and Efficient Middleware," in *International Conference on Distributed Computer Systems (ICDCS)*, 2003. Extended version available from `http://www.cc.gatech.edu/~yannis`.

[89] Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis, and Marc Fleury, “Aspectizing Server-Side Distribution,” in *Automated Software Engineering (ASE)*, Montreal, October 2003.

[90] Eli Tilevich and Yannis Smaragdakis, "Portable and Efficient Distributed Threads for Java", in *ACM/IFIP/USENIX 5th International Middleware Conference (Middleware)*, 2004.

[91] Eli Tilevich, Yannis Smaragdakis, and Marcus Handte, “Appletizing: Running Legacy Java Code Remotely From a Web Browser,” *IEEE International Conference on Software Maintenance (ICSM)*, 2005.

[92] Eli Tilevich and Yannis Smaragdakis, “Binary Refactoring: Improving Code Behind the Scenes,” *International Conference on Software Engineering (ICSE)*, 2005.

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

[94] Ronald Veldema, Rutger F.H. Hofman, Raoul A.F. Bhoedjang, and Henri E. Bal, “Runtime Optimizations for a Java DSM Implementation,” in *Joint ACM JavaGrande - ISCOPE 2001 Conference*, June 2001.

[95] Ronald Veldema, Rutger F.H. Hofman, Raoul A.F. Bhoedjang, Ceriel J.H. Jacobs, and Henri E. Bal, “Jackal: A Compiler-Supported Distributed Shared Memory Implementation of Java,” in *Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'01)*, June 2001. Also under the title: Source-Level Global Optimizations for Fine- Grain Distributed Shared Memory Systems.

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

[97] Mark Weiser, “The Computer for the 21st Century,” *Scientific American*, 265(3):94-104, Sep. 1991.

[98] Danny Weyns, Eddy Truyen, and Pierre Verbaeten, “Distributed Threads in Java,” in *The International Symposium on Distributed and Parallel Computing (ISDPC)*, 2002.

[99] Danny Weyns, Eddy Truyen and Pierre Verbaeten, “Serialization of Distributed Threads in Java,” *Special Issue of the International Journal on Parallel and Distributed Computing Practice, (PDCP)*, vol. 6(1), 2004.

[100] Paul R. Wilson, “Uniprocessor Garbage Collection Techniques,” in *International Workshop on Memory Management*, St. Malo, France, September 1992.

[101] Ann Wollrath, Roger Riggs, and Jim Waldo, “A Distributed Object Model for the Java System,” in *USENIX 1996 Conference on Object-Oriented Technologies*, pages 219-232, Toronto, Ontario, Canada, June 1996.

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

[103] XDoclet, `http://xdoclet.codehaus.org/` .

[104] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers, "Untrusted Hosts and Confidentiality: Secure Program Partitioning," *the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[105] Dong Zhou, Santosh Pande, Karsten Schwan, "Method Partitioning - Runtime Customization of Pervasive Programs without Design-time Application Knowledge," in *International Conference on Distributed Computer Systems (ICDCS)*, 2003.

[106] Dong Zhou and Karsten Schwan, ``JECho - Supporting Distributed High Performance Applications with Java Event Channels," *International Parallel and Distributed Processing Symposium (IPDPS)*, May 2001.

[107] David Zook, Shan Shan Huang, and Yannis Smaragdakis, “Generating AspectJ Programs with Meta-AspectJ,” *Generative Programming and Component Engineering Conference (GPCE)*, 2004 .

VITA

Eli Tilevich was born in Leningrad, USSR, and moved to the United States with his family before his place of birth became St. Petersburg, Russia. Until his mid-twenties, he was a professional classical musician, teaching privately and playing clarinet as a free-lancer in the New York City area. In the Spring of 1994, he broke his wrist and could not play for several months. The incident diverted the upward trajectory of his career path, and the inability to play impelled him to explore professional opportunities other than music. Soon after, while taking computer science courses at Pace University, he discovered an aptitude and enthusiasm for computer programming. After graduating from Pace University *Summa Cum Laude* in 1997, he was offered a software developer's position at Information Builders Inc., the oldest and largest software development company in New York City. Seeking to broaden and deepen his knowledge and expertise in computing along with building his professional career, he enrolled in the graduate program in Information Systems at New York University. Upon graduation from NYU with an M.S. in 1999, he decided to pursue his Ph.D. at the Georgia Institute of Technology. He pursued his studies under the direction of Dr. Yannis Smaragdakis, focusing his research on the systems and programming languages end of software engineering. His research publications have appeared in venues including the European Conference on Object-Oriented Programming, the International Conference on Software Engineering, IEEE Pervasive Computing, and others. Upon graduation from Georgia Tech in December of 2005, he accepted a faculty position with the Department of Computer Science at Virginia Tech.