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 network" 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 reproduced 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.