

# StratOSphere: Mobile Processing of Distributed Objects in Java\*

Daniel Wu Divyakant Agrawal Amr El Abbadi  
Department of Computer Science  
University of California, Santa Barbara  
{danielw, agrawal, amr}@cs.ucsb.edu

## Abstract

We describe the design and implementation of our **StratOSphere** project, a framework which unifies distributed objects and mobile code applications. We begin by first examining different mobile code paradigms that distribute processing of code and data resource components across a network. After analyzing these paradigms, and presenting a lattice of functionality, we then develop a layered architecture for **StratOSphere**, incorporating higher levels of mobility and interoperability at each successive layer. In our design, we provide an object model that permits objects to migrate to different sites, select among different method implementations, and provide new methods and behavior. We describe how we build new semantics in each software layer, and finally, we present sample objects developed for **StratOSphere**.

## 1 Introduction

The advent of distributed computing has had a major influence in the computing industry in recent years, witnessed by the growth of mobile computers and networked computing systems. The desire to share resources, to parcel out computing tasks among several different hosts, and to place applications on machines most suitable to their needs has led to distributed programming systems such as CORBA[Sie96] and DCOM[Box97] that predominate in the marketplace. Despite competing standards, both systems have very similar designs. They each define a distributed object model that achieves interoperability through strict separation between interface and implementation. A CORBA or DCOM object registers its IDL (Interface Definition Language) interface into an Interface Repository. The IDL interface specifies method signatures for each object, describing in part its behavior and semantics. Applications then obtain references to each distributed object, and invoke operations based upon these method signatures.

\*Work supported by research grants from NSF/ARPA/NASA IRI9411330 and NSF CDA-9421978.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MOBICOM 98 Dallas Texas USA

Copyright ACM 1998 1-58113-035-x/98/10...\$5.00

The benefits of this approach are evident: These systems provide location transparency, so that applications residing on different hosts may invoke one another; the object model in each system promotes encapsulation of program code so as to facilitate management of remote objects; the separation between IDL interface and code implementation permits implementations in different languages through different language bindings<sup>1</sup>. The drawback, however, is that mobile code applications are not well supported. As of yet, there exist no standardized services in either CORBA or DCOM to migrate objects and threads for distributing processing, no facilities for customizing or providing new object behavior at run-time, no provisions for agent processing or autonomous computing. Although these features can certainly be grafted on top of CORBA and DCOM systems, doing so would add an even more cumbersome layer of services onto—especially in the case of CORBA—an already bulky software package.

In seeking to support mobile code applications, we take a different approach: We first examine the requirements of mobile code systems, and develop a corresponding software system with higher-level abstractions of *code*, *data*, and *migration approaches*. To achieve this result, we describe the design of the **StratOSphere** architecture, which supports mobile applications written in Java. Every entity in **StratOSphere** is modeled as a Java object. In the Alexandria Digital Library (ADL) project at UC Santa Barbara [SF95], for example, the library's collection of high resolution aerial photographs, satellite images, and topographical maps would each be modeled as an object, replete with methods to filter, layer, and display the object instance. The library's current holdings consist of over 700,000 catalogue items, with some images exceeding a gigabyte of storage. The design of a digital library for this collection requires that the vast amounts of data be distributed across different hosts. Furthermore, not only must the data be distributed, but the processing of the data must also be distributed. To address these issues, we have developed the **StratOSphere** system that permits both object instances and object methods to be migrated to different sites for remote execution. In our design, we provide objects with the capability of obtaining new functionality by dynamically locating and processing new executable code.

The paper is organized as follows: Section 2 describes related work on distributed programming; Section 3 describes

<sup>1</sup>Indeed, Xerox PARC's variant of CORBA, ILU[JS00], mixed-language programming is supported to such an extent that ILU objects implemented in different languages can reside in the same process space.

the requirements of distributed object management; Section 4 describes the design and implementation of StratOSphere; Section 5 concludes the paper.

## 2 Related Work

Work in distributed objects and interoperability has had a long history, beginning with attempts to extend programming to remote hosts in order to provide greater collaboration and sharing of resources. These efforts can be classified into three basic categories: distributed programming libraries and packages, distributed operating systems[Gos91], and distributed programming languages. These categories reflect attempts to extend a programming environment by adding a distribution layer to an existing language, by providing distributed access as an operating system primitive, and by developing a language with fully distributed scope and semantics. These designs each have their various advantages and disadvantages, depending on the amount of programming effort required upon the application developer, the amount of specialized infrastructure needed to provide an interoperability layer, and the ease of extending an existing programming system.

### 2.1 Distributed Programming Systems

Distributed programming systems provide the simplest form of distributed programming support, by adding a distribution layer on top of a language system. They usually provide remote stubs at each end of hosts that wish to communicate and perform remote invocations. Systems such as RPC[BN84], DCE[Fou92], CORBA[Sie96], DCOM[Box97], RMI[WRW96], and HORB[Sat96] each generate proxies to provide distributed access to remote resources. While RMI and HORB are Java-specific, RPC, DCE, CORBA, and DCOM provide language bindings for a particular language implementation. In doing so, they are able to unify different language systems to build common applications. Their main advantage is that they require relatively little effort to incorporate into an existing software system; their disadvantage is that the limited form of interoperability and object migration that they support, although CORBA, DCE, and DCOM seek to ameliorate this by providing a rich set of standard services for creating, locating, registering, and storing an object<sup>2</sup>.

### 2.2 Distributed Operating Systems

The operating system approach addresses a different aspect of distribution by migrating processes for load-balancing, fault-tolerance, and resilience. Systems such as Sprite[DO87], V[Che88], and Locus[PE86] provide built-in operating system support to freeze the run-time computation of a process, migrate the process to a remote site, and unfreeze the process. Each migration entails leaving a proxy behind to forward I/O and communications to the new site. By providing fine-grained mobility, the location of processors, memory, and file systems can be made fully transparent to an application program. Though providing an extremely powerful and elaborate design, users would have to upgrade and install an entirely new operating system to their existing computer system—a prohibitive modification in current enterprises.

<sup>2</sup>See CORBA's Common Object Services Specification[Sie96], DCE's object services[RKF92], and DCOM library services[Box97] for further details.

## 2.3 Distributed Languages

Among distributed languages, there are languages designed with fully distributed semantics such as Obliq[Car95], Emerald[JLHB88], and Distributed Oz[vRHea97], and adaptive mobile code languages such as TACOMA[JvRS95], Telescript[Whi96b], Sumatra[ARS97], Agent TCL[Gra96], Odyssey[Whi96a], Voyager[Obj96], and the Liquid Software project at University of Arizona[HMPP96]. In the former category are languages whose semantics were designed to support distributed scope and access, provide a shared memory abstraction, and extend ordinary language operations to manage replicated objects and data. The latter category comprise agent languages and mobile code frameworks which provide object mobility within a distributed environment.

The Liquid Software project, in particular, introduces the notion of dynamically moving functionality within a network to enable adaptive computing. Consisting of an array of retargetable compilers, customizable client/server interfaces, and operating system support library, the Liquid Software infrastructure can be used to develop software agents for browsing, searching, and retrieval. In StratOSphere, we take a very similar approach by developing a distributed object system that provides network transparency for the location of data and the location of its execution; in StratOSphere, however, we provide an *object repository* that resides at each site where processing takes place. We show how remote object repositories consisting of instances and methods permit objects to take on dynamic functionality by acquiring new methods and behavior at these remote sites.

## 3 Mobile Processing and Mobile Objects Requirements

Before describing the architecture of StratOSphere, we first examine the forms of mobility provided by various distributed and network programming packages. We analyze different execution scenarios to determine what form of mobility is required of the distributed system.

### 3.1 Mobile Code Paradigms

Carzaniga, Picco, and Vigna[CPV97, CGP<sup>+</sup>97] provide an elegant description of several mobile code design paradigms for distributed applications. These are classified as *Client/Server* (CS), *Remote Evaluation* (REV), *Code on Demand* (COD), and *Mobile Agent* (MA) paradigms. By decomposing distributed applications into *resource components* (code and data), *computation components* (thread of execution), *interactions* (event and information passing between two or more components), *sites* (location where processing takes place), distributed execution can be modeled as primitives operating in one of the above mobile code scenarios.

We briefly describe each mobile code presentation as follows: Computation components A and B reside at sites  $S_A$  and  $S_B$ , and component A initiates some interaction with component B. Code and data resources from components A and B, ( $C_A, D_A, C_B, D_B$ ) are presented below before and after each migration in Table 1.

In the CS paradigm, the code, data, and execution remain fixed at the server site  $S_B$ . This is the usual RPC style of programming in which a client requests a service of a server by specifying some data resource  $D_A$ . The program code  $C_B$  and remaining data resources  $D_B$  to perform this service are resident with the server. An additional interaction from B to A returns the result of the execution.

Paradigm	Before Migration		After Migration	
	$S_A$	$S_B$	$S_A$	$S_B$
CS	$A, D_A, C_A$	$B, D_B, C_B$	$A, D_A, C_A$	$B, D_B, C_B, D_A$
REV	$A, D_A, C_A$	$B, D_B, C_B$	$A, D_A, C_A$	$B, D_B, C_B, D_A, C_A$
COD	$A, D_A, C_A$	$B, D_B, C_B$	$A, D_A, C_A, C_B, D_B$	$B, D_B, C_B$
MA	$A, D_A, C_A$	$- D_B, C_B$	$- D_A, C_A$	$A, D_B, C_B, D_A, C_A$

Table 1: Mobility and remote execution paradigms

In the REV scenario, the code to perform the execution is stored at  $S_A$ . Component A ships both code and data to site  $S_B$  where it gets processed at  $S_B$ . A final interaction returns the result from B to A. Although mobile code scenarios CS and REV are very similar in nature, they differ in one key aspect: the ability to transfer processing from one site to another by transferring the executable code. In languages such as LISP, Scheme, or any of a variety of scripting languages, REV can be seen as a natural extension of CS, since data (represented as lists, strings, or tuples) can just as easily be interpreted as code. That is,  $C_A$  can be represented as  $D_A$ . As long as a program interpreter is present at each server site, remote evaluation and execution are readily available. For procedural languages which distinguishes between programming instructions and data structures, however, the run-time requirements of REV differ markedly from those of CS. The additional requirements of shipping code from one site to another affects the environment's architecture and implementation semantics.

Scenario COD is an inversion of REV. Instead of initiator A sending code and data off to B, component A requests code and data from B, and executes it locally. An example of this form of code mobility is the applet service in the Java programming language, in which HTTP browsers download Java code from remote sites for local execution. An additional service known as *servlet* programming[Cha97] performs the opposite service, pushing Java code from a local client to a remote server for remote execution, thus falling under the REV category. While COD and REV are very similar in design, the performance differences between the two often lead to one scenario being preferred over the other. In Web computing, the applet model COD proliferates, as the bulk of processing requires that computations be offloaded to client sites. However, in more complex interactions, servlets and CGI[Gun96] programming perform execution at the server-side, where relevant data and resources are stored.

Finally, MA, is the most dynamic and autonomous of the above paradigms. In the three previous scenarios, code and data components can be transferred from one site to another to re-direct the location of processing, but the computation component (the executing process) remains fixed to its original site. In CS, REV, and COD, initiator A begins an interaction and obtains the result in a separate thread from that of the receiving component B. In MA, however, not only can resource components be transferred, but so can the entire computational component. Indeed, from Table 1, we observe that MA is an extension of REV, in which not just  $D_A$  and  $C_A$  are migrated, but also the computational component A itself. To perform this service, the mobile agent A suspends its execution at site  $S_A$ , preserves its state into an *execution unit* (EU), and transfers the EU to site  $S_B$  where processing resumes from A's recent state.

These four mobile code paradigms are among the types

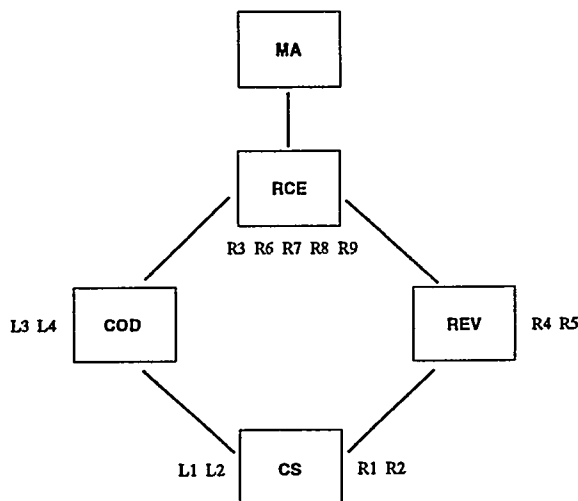


Figure 1: Lattice of functionality in mobile code paradigms

of mobility we support in StratOSphere. We introduce another mobile code paradigm called RCE (Remote Code Execution), which is a union of REV and COD mobility; a RCE facility is able to transfer code  $C_A$  and data  $C_A$  to a remote site for remote execution, as well as pull remote code  $C_B$  and data  $D_B$  for local execution. They can be organized into the following lattice (Figure 1) to evaluate the mobile computing capability of each scenario. The least functional mobile code paradigm is CS, in which both code and computational components are fixed, though data components are migratory. In REV, COD, and RCE scenarios, both code components and data components are mobile but the computational components remain fixed. All three components are mobile in the most functional mobile code paradigm of all, MA. In principle, it may appear that MA is the only mobile code paradigm required, as it provides migration of all computational components. Indeed, mobile agent technologies advance this supposition. The following example, though, shows why MA may not suffice.

Suppose code and data components  $C_A$  and  $D_A$  are stored at site  $S_A$  while data component  $D_B$  is stored at site  $S_B$ , but execution is to take place at site  $S_C$  (Figure 2); assume further that the initiating process resides at site  $S_A$ . Adopting a sole MA approach, the execution unit may load in  $C_A$  and  $D_A$ , then visit site  $S_B$  to load in  $D_B$  (Figure 2a), before finally arriving at  $S_C$  to execute  $C_A$  with  $D_A$  and  $D_B$ . The drawback to this approach is that components  $C_A$  and  $D_A$  are migrated twice: from  $S_A$  to  $S_B$ , then from  $S_B$  to  $S_C$ ; if the data item  $D_A$  is very large then the performance penalty that MA incurs may be prohibitive.

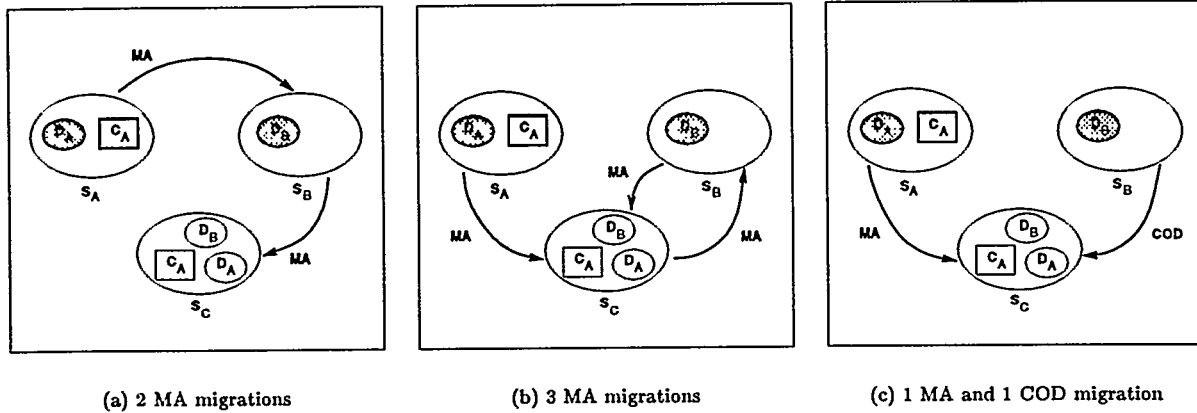


Figure 2: MA and MA/COD migrations.

To alleviate the costs of transferring large data components, MA may be employed in a different manner. After loading in  $C_A$  and  $D_A$ , the execution unit first visits  $S_C$  and deposits  $C_A$  and  $D_A$  in a repository at  $S_C$ ; next the execution unit migrates to  $S_B$  where it collects  $D_B$  and returns to  $S_C$  where execution can proceed with code and data components are in place. This scenario (Figure 2b) requires three migrations of MA. While the overhead of unnecessarily transferring large data objects has been eliminated, this approach also incurs a performance penalty, albeit less evident than the preceding scenario. In depositing the components  $C_A$  and  $D_A$  at  $S_A$ , then migrating the EU to  $S_B$  and back to fetch  $D_B$ , the EU must initially unload components  $C_A$  and  $D_A$  into a repository at  $S_B$  and then reload those same components before execution can proceed. We note further the two MA migrations to  $S_B$  and back are for the purpose of fetching components, a feature for which the COD mechanism was designed. By replacing these two MA migrations with a single COD transfer (Figure 2c), the execution unit can thus avoid having to insert and subsequently retrieve  $C_A$  and  $D_A$  from storage at  $S_A$ . Thus, though MA provides a highly desirable form of mobility, the remaining forms of mobility cannot be discarded. In designing StratOSphere, we provide a general migration facility that supports all the above forms of mobility, in order to adapt to a number of different execution environments.

### 3.2 Execution Scenarios

Expanding upon the previous example, we now consider the possible execution scenarios where code and data reside at different locations, each requiring a separate transfer to the execution site. As Semeczko and Su[SS97] have pointed out, there are 13 possible data and execution location scenarios for data and executable programs stored at local and remote hosts. In these scenarios, both the data and executable can be migrated to different hosts for remote processing. Table 2 lists the scenarios for *local execution*, while Table 3 lists the scenarios for *remote execution*. In these tables,  $L$  represents the local host, and  $R$  the remote host, while  $E$  represents the host where execution is to take place.

The four local execution scenarios (L1–L4) depict those scenarios where code and data are to be pulled to the local site for local execution. They require a COD facility to satisfy their mobility requirements. Of these scenarios, L1

	Location of Code	Location of Data	Mobile Paradigm
L1	L	L	CS
L2	L	R	CS
L3	R	L	COD
L4	R	R	COD

Table 2: Local execution scenarios

	Location of Code	Location of Data	Mobile Paradigm
R1	E	E	CS
R2	E	L	CS
R3	E	R	RCE
R4	L	E	REV
R5	L	L	REV
R6	L	R	RCE
R7	R	E	RCE
R8	R	L	RCE
R9	R	R	RCE

Table 3: Remote execution scenarios

and L2 require only CS for data passing, which is readily achieved through client/server distributed computing packages such as CORBA, DCE, and RPC.

The remote execution scenarios, however, require a greater degree of mobility and functionality. Since the component initiating the processing does not reside at the execution site, a COD mechanism does not suffice. For these remote executions, four of the cases (R1, R2, R4, R5) can be implemented through the use of REV frameworks, of which two (R1 and R2) require only CS. For the remaining five cases, we proceed higher up the mobility ordering to obtain RCE functionality. In each of the cases (R3, R6, R7, R8, R9), a RCE facility can be employed to visit first the execution site, and from there pull any remaining code and data from a remote site to the execution site for processing.

Altogether, these 13 scenarios comprise the possible combinations of executions in a distributed environment, where code and data can both be migrated for mobile code applications. Supporting such functionality, however, is only one aspect of a distributed object system; mobility serves distribute processing throughout a network, but we need further facilities to manage the creation and execution of mobile objects.

### 3.3 External Object Methods

In StratOSphere, an object's methods may be stored separately from the object instance. We view this as a fundamental requirement in permitting objects to adapt to changing conditions in a fluid distributed environment where changes in operating conditions are inevitable. By permitting an object's methods to be substituted at run-time, an object can modify and extend its services without having to recompile.

At first glance, this treatment may appear surprising to adherents of object-oriented programming, as an object is traditionally understood to store both data (object state), and executable code (object method) as a single programming unit; thus, decoupling an object's state and methods seems contrary to an object-based perspective. On the one hand, the demands of mobile and distributed computing environments promote the separation of large object data and object services, while on the other hand, the object-oriented approach requires their co-location. To reconcile these two views, we model each object as a loosely-coupled object. That is, an object consists of *internal methods* that are tied to an object, as well as *external methods* that operate on the object state, but are stored externally. In programming terms, an internal method is a method that resides in a class, while an external method resides in a separate executable program as a Java .class file. Internal methods are those methods that provide consistent modification of object state, while external methods provide extended services. At run-time, a Java object's internal method may be invoked by means of the Reflections package[Sun97], while an external method can run upon an object by using a Java class loading mechanism[GJS96]. We explore these concepts in greater detail below.

#### 3.3.1 Run-Time Object Adaptation

Beyond resource and computation migration, the ability to adapt a running object to changing needs and requirements remains an essential aspect of an interoperable object system. For this we require the ability to extend the services of an object. For example, consider a scenario in which a client wishes to process an aerial image of a region to determine areas of vegetation growth. The StratOSphere ob-

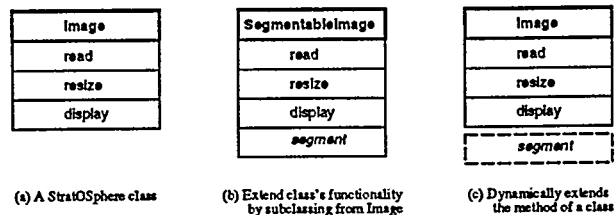


Figure 3: Extending a method in StratOSphere

ject in the system, called Image, contains methods to read in an image, resize, and display the image. The client, furthermore, would like to segment the aerial image using image processing routines to highlight the areas of vegetation growth.

In traditional object systems, the client would usually subclass from the Image class in order to extend its functionality by providing a segment method. Creating a new subclass, however, is a compile-time activity; the client declares a new class, SegmentableImage, that inherits from the base class, adding and redefining new methods and instance variables as appropriate. This technique works well in object-oriented languages, because the programmer can freely extend classes and introduce new types and features into the programming environment. This capability, however, cannot be sustained in distributed environments, where each user is limited in the ability to add or modify system classes. Rather than provide a new subclass at compile-time, the StratOSphere system lets users supply a new method to an object at run-time (Figure 3).

In many environments, security restrictions forbid a client from arbitrarily introducing a new type into the system. For applications, however, this inability to modify or extend the services of an object, poses too rigid and unwieldy a restriction. As time progresses, the role of an object evolves and adapts[RS91], and a software system must accommodate these changes by providing the necessary user enhancements and modifications. For this reason, we too must provide a means of dynamically extending an object's services in our StratOSphere system. In doing so, we must ensure that redefining or adding a new method does not lead to an inconsistent type system for our objects. As we shall see, the means by which we ensure consistency is through run-time assertions of methods and class invariants.

#### 3.3.2 Repository Re-Implementation

Taking this method one step further, we not only permit an applications to extend its object behavior, we provide repositories with this capability as well. Specifically, we let object repositories provide different implementations of external methods.

Suppose after some time, a SegmentableImage class gets introduced into the StratOSphere environment (Figure 4a). A client application may still not be satisfied with the given segment method. Once again, the client application can provide its own implementation by dynamically extending the class and redefining the segment operation, or better yet, the client can shop around the network and locate an already existing repository method to redefine the operation (Figure 4b).

In this case, we let the repository rather than the object

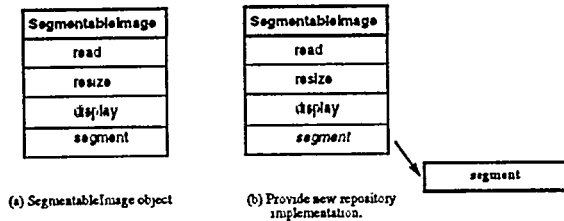


Figure 4: Repository redefines method with new implementation

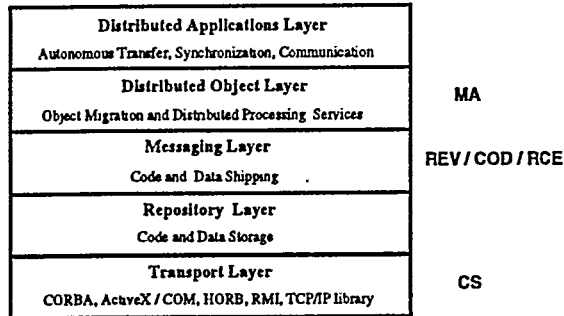


Figure 5: StratOSphere Design Layers

reimplement a method in order to obtain new functionality or take advantage of a particular host repository's resources. By letting both applications and repositories specialize their services, the StratOSphere environment provides convenient adaptation for both source program and distributed environment, presenting a customizable run-time environment to adjust to changing conditions. We shall see how we obtain these service when we describe the architecture of the StratOSphere system.

#### 4 StratOSphere Architecture

Figure 5, illustrates the multi-tiered layer of design services that comprise StratOSphere. These layers consist of the Transport layer, the Repository Layer, the Messaging Layer, the Object Migration Layer, and the Distributed Application Layer. Annotated next to each relevant layer is the form of mobility provided by the architecture.

##### 4.1 Transport Layer

The bottommost layer of StratOSphere is the Transport Layer which provides basic remote invocation services for distributed applications. In this layer we take a simple CS system as a substrate upon which we can provide further mobility paradigms. Here we may select from the myriad of packages: RPC, CORBA, DCOM, DCE, and Java-specific interoperability frameworks such as HORB[Sat96] or RMI[WRW96]. Each of these packages permit a client to pass arguments and invoke a method of a remote object, wait for the method to be executed at the server and obtain a result, thus satisfying basic CS functionality.

For the Transport Layer, we have chosen HORB as our CS package to provide remote access and remote execution.

Though HORB is capable of much greater functionality, we utilize HORB simply to provide delivery and remote access to objects. Access to these HORB services are restricted to the Transport Layer, so that transport mechanisms can be substituted in place without disturbing the remaining layers of the StratOSphere architecture; we could just as well have replaced HORB with any of the other packages above by re-implementing our Transport Layer interface. Indeed, General Magic's Java-based agent and distributed object system, Odyssey[Whi96a], lets the application programmer select among RMI, DCOM, and CORBA IIOP for its transport.

##### 4.2 Repository Layer

At each site in a distributed system, we require the services of a repository to store data and code, where data and code correspond to *instances* and *external methods* of a StratOSphere object. The Java object serializer[RWW96] is employed to serialize each object into a byte stream for storage in a repository. Since an external method is a Java .class file, it can be stored in a repository as a stream of Java bytecode. In this way, the repository layer serves as a persistent storage system for mobile objects in StratOSphere. By storing serialized instances, we can record the state of an object, as it proceeds in computation. A repository further serves as a library of services by storing external methods that act upon object instances.

To operate within the StratOSphere framework, data instances and external methods must satisfy the following Java interfaces:

```
public interface SSObject extends Serializable {
}

public interface SSInstance extends SSObject {
    public boolean invariant();
}

public interface SSMMethod extends SSObject {
    public boolean precondition();

    public SSResult run(SSInstance instance,
        String methodName,
        SSArglist arglist) throws SSError;

    public boolean postcondition();
}
```

Each StratOSphere instance implements the SSInstance interface in order to be serialized and stored in a repository, while each external method implements SSMMethod in order to be dispatched by the StratOSphere's run-time environment. We defer discussion of the *invariant*, *precondition*, and *postcondition* methods in these interfaces until Section 4.4; for now, we will examine the *run* method in the SSMMethod interface, which is used to dispatch an external method upon an instance.

The implementation of the method dispatcher is described in [WAAS97]. Basically, StratOSphere employs the MSQl database[Hug95] to store instance, class hierarchy, and class format meta-data tables. We wrote a new classloader[GJS96] that queries these tables to locate the appropriate internal or external method to apply upon an object, based upon the name of the method specified in *methodName*, and the instance class type. If the query resolves to an external

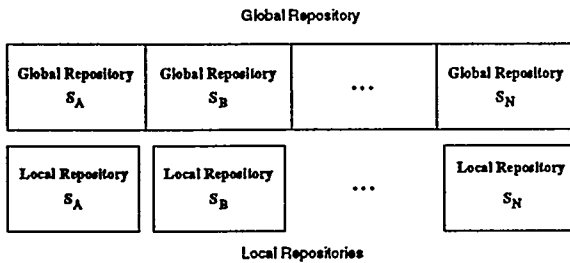


Figure 6: StratOSphere Repository Structure

method, then the dispatcher uses our classloader to load the external method from the repository, and invoke the run method, passing in the required arguments: the StratOSphere instance, the name of the method, and an appropriate argument list. If the query returns an internal method of the StratOSphere instance, then we employ the Java Reflections Library[Sun97] to dynamically invoke the internal method of the object instance, again passing in the relevant arguments: the object instance and argument list.

#### 4.2.1 Repository Structure

Figure 6 illustrates our repository structure within StratOSphere. Each site implements both a *Local repository* and a portion of a *Global Repository*. The Global Repository can be viewed as a global address space partitioned among the hosts in the network; objects stored in the Global Repository are visible at every StratOSphere site. Each Local Repository, however, is separate from the others. A Local Repository caches object instances, and provides local implementations of external methods. A discussion on how we maintain consistency among different copies of objects cached at each Local Repository will not be presented in this paper, but we briefly describe the storage of external methods at each Local Repository.

As noted above, our classloader queries a set of metadata tables at each repository site to locate a particular method to dispatch. Each repository site may, in fact, provide a different implementation of an external method. For example, sites  $S_A$  and  $S_B$  may each store a different implementation of the *segment* method in their Local Repositories. A Image object visiting  $S_A$  would obtain a different behavior than if it were to visit site  $S_B$ . Our StratOSphere framework thus supports the *factory pattern*[GRHV95], in which a common abstract interface is adopted to obtain different concrete method implementations at each site. A repository site can take advantage of this feature to provide specialized behavior for a particular method. A different segmentation algorithm may be implemented for the *segment* method, for example, if the host were a supercomputer as opposed to a desktop machine.

In the StratOSphere system, our repository structure exhibits both vertical and horizontal partitioning. The Global Repository is vertically partitioned to extend a common storage space across different sites. The Local Repositories are horizontally partitioned to implement new object behavior at each individual site.

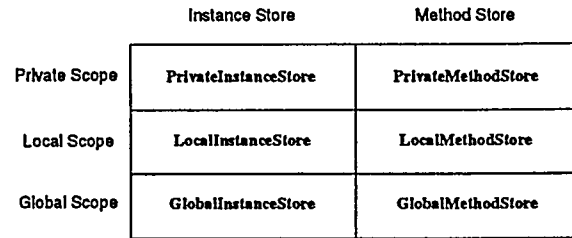


Figure 7: StratOSphere Repository Schema

#### 4.2.2 Repository Scope

The schema for our object repository is shown in Figure 7, where we note that instance and method objects each have three levels of scoping: *Global*, *Local*, and *Private*. Global scope is extended to elements in the Global Repository, as it serves as a global address space for the entire network. A site with a Local Repository has Local scope, since its implementations are visible only to itself, and not to any other site. Finally, we come to the concept of Private scope, which is applicable to MA applications. Each StratOSphere mobile agent contains a Private Repository that travels with the agent as it visits different sites. We will discuss this notion of Private Repository further in Section 4.4, when we describe our MA implementation. There we shall see that a mobile StratOSphere application can compose different views of a distributed object by visiting different repositories for their local implementations, yet retain access to Private and Global repository objects.

#### 4.3 Messaging Layer

The next layer of our design is the Messaging Layer, in which we model interactions as message-passing activity between remote entities. A sender sends a message to a receiver by specifying a remote repository as a location. The message is then delivered to its destination through the Transport Layer using the Repository Layer to reference the target repository.

In this layer, each message passed between remote objects is itself a Java object, and not simply a coded tag or field value. For this layer, we provide an interface called *SSDispatchable*.

```
public interface SSDispatchable {
    public SSResult dispatch() throws SSEException;
}
```

Each message sent from client to server corresponds to a Java object that implements this interface, by defining a *dispatch* routine. Upon receiving the message, the server simply calls *dispatch* to process the message. Since the message is an object, the code to execute the message is encapsulated entirely in the message, and need not be registered at the server.

By designing the message-passing mechanism in this manner, we can implement REV and COD functionality for our mobile code system. As noted earlier, REV can be viewed as a natural extension of CS, if executable code were able to be represented as data; in the case of Java this poses no barrier, as external methods in the form of Java programs can be stored as an array of Java byte-code. The byte code can then be inserted into a message along with the serialized object

instance, and passed from client to server. When the server processes the message, the executable byte-code is subsequently extracted from the message and executed upon the instance using a `StratOSphere` Java classloader[GJS96]. The result of the execution is sent back to the client by means of a subsequent message.

By factoring out the message-passing interaction between client and server in CS and REV, we are able to build REV functionality on top of the Transport layer's CS paradigm. Likewise, COD can also be implemented in a very similar manner: A message is sent to a server requesting a repository method; the server obtains the byte-code for the external method and passes it back to the client a return message; the client extracts byte-code, dynamically loads and executes the method at the client's site. The manner in which we obtain both REV and COD functionality is through specialization of the messages that are passed between client and server.

In addition to implementing a `Dispatchable` interface, a message object must provide one other general routine, which is the `send` method:

```
public abstract class Message implements SDispatchable {
    public SSRresult send() throws SException {
        ...
    }
}
```

The `send` method transfers the message object to a given repository site. From this message-delivering abstract Java class, we may then derive other subclasses to provide specialized services.

In Figure 8, we see the pertinent message objects in the `StratOSphere` system, with the `Message` at the root of this hierarchy. A client sends a `QueryMessage` to locate and explore information at remote sites. An `AdminMessage` is issued by a privileged client to perform some administrative function such as shutting down the repository site, introducing a new object type into the system, or modifying the contents of an object repository. The `RemoteEvaluationMessages`, `MobileObjectMessages`, and `RemoteExecutionMessages` provide the system with mobile code functionality. Each of these messages holds an `SSRepositoryObject` to fetch or store data instances and external methods into the repository. The `RemoteEvaluationMessage` first fetches data and code from the local repository and delivers the message to the destination site for remote execution, providing the system with REV capability. To satisfy COD, the `MobileObjectMessage` is sent to fetch objects from the remote repository. Finally, the `RemoteExecutionMessage` provides RCE functionality by sending the message to the execution site, then issuing `MobileObjectMessage` to pull the relevant code and data and directly to that site for processing.

The Messaging Layer accesses the services of the Repository Layer through a set of well-known interfaces. To obtain an object in a `StratOSphere` repository, these messages go through the `SSRepositoryObject` interface,

```
public interface SSRepositoryObject {
    public void bind() throws SException;
    public void fetch() throws SException;
    public void store() throws SException;
}
```

which defines `bind`, `fetch`, and `store` methods<sup>3</sup> The `bind` method associates a particular code or data item with a

<sup>3</sup>For sake of simplicity, formal arguments of methods will not be shown in the text.

repository; the `bind` method must always be called before any further operations can be invoked. The `fetch` method is called to extract the item out of repository storage, while the `store` method inserts it back in.

We note that in each of these three messages that the order in which the `send` of the message and the `fetch` of the `SSRepositoryObject` vary. The `RemoteEvaluationMessage` performs a `fetch` of the objects before the `send` to the server site. Conversely, the client must first `send` the `MobileObjectMessage` to the repository site, before it can `fetch` the repository item and return it to the client. Since RCE is a union of REV and COD, the `RemoteExecutionMessage` will first `fetch` any local data before the `send` to the execution site, whereupon the `RemoteExecutionMessage` issues a `MobileObjectMessage` to `fetch` any remaining objects from remote repositories.

With REV, COD, and RCE in place in the Messaging Layer, the `StratOSphere` system can satisfy both local and remote execution scenarios: L1-L4 and R1-R9. To obtain MA functionality, however, we must develop further mechanisms at higher levels of `StratOSphere`.

#### 4.4 Distributed Object Layer

The previous Messaging Layer provided a means of shipping code and object instances across remote sites in order to re-target processing of the object. In this layer, we use the services of the Messaging Layer to implement a distributed object, a mobile object that can be dynamically relocated to a different host, in a type-safe manner. To do so, we introduce the addition of a global naming service and a distributed execution facility.

##### 4.4.1 Global Naming Service

The Messaging Layer provides passing of objects and methods so that objects are copied to remote sites, but to provide distributed processing, we must also have some means of accessing object by reference. For this, we require the addition of a global naming service that keeps track of the location of each object as it is migrated, and maintains a global pointer to the running object.

In our current design for `StratOSphere`, we employ the services of a centralized database to track object state and location. This database is considered to be stored within the global repository. Each time a `StratOSphere` instance is created, an entry is added into a table, providing the following information:

(ObjectName, ObjectRef, Host, Owner, Type, State, Time).

The *ObjectName* is a string assigned to an object, and used as a primary key for this global name table. The *ObjectRef* is an unsigned integer value that stores a host-specific reference to the object. The *Owner* and *Host* fields describe the hosts where the object was originally created, and where it is currently located. The *Type* field stores the qualified name (full package name) of the Java object, while *State* value describes the current object state (*Created*, *Transferred*, *Stored*, *Running*, *Destroyed*), corresponding to the states when the object was created, when it was migrated to a new site, when it was checked in and checked out of a persistent store, when an operation was invoked on the object, and when it was destroyed or garbage collected. Finally, the last field *Time* records the global timestamp that the entry was entered into the database.

Using this information, we can keep track of an object's current status and provide access to an object through a

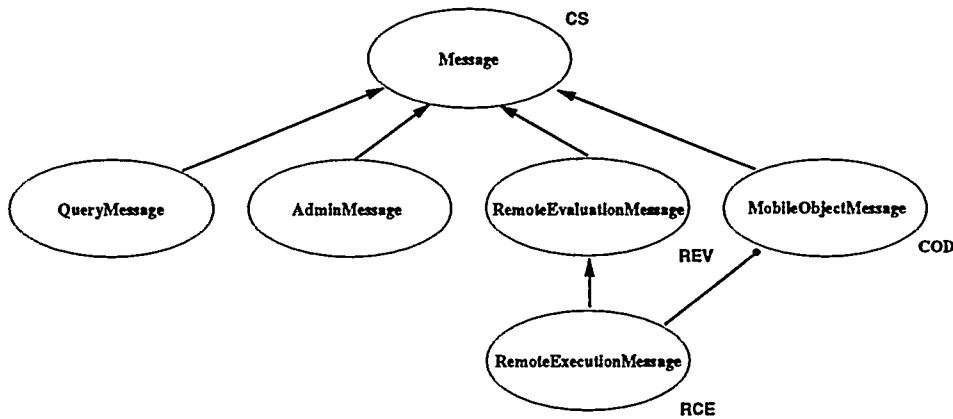


Figure 8: StratOSphere Messages

global reference. Though our approach is currently based upon a centralized server, we have studied plans to redesign the naming server into a distributed server. The work of Awerbach and Peleg[AP95], for example, describes the design of a distributed directory for mobile users, that tracks the location of each mobile object in an efficient manner by incrementally propagating updates into the directory for specific subnetwork regions.

#### 4.4.2 MA Facility

The next service that we provide is a means of distributing the execution of an object throughout the network: MA. In the Messaging Layer, we designed a facility to migrate and deliver objects by storing them within appropriate messages, and passing the messages between hosts. The drawback with this technique is that the object migration is specified on a peer-to-peer basis; there is no coordination among different hosts to perform distributed processing of an object. The following example illustrates the need for this facility to distribute execution:

Suppose we have a client application that is logged on a laptop machine named *skinny*. The client would like to create a *LandsatImage* of Santa Barbara County, segment the image to locate areas of vegetation growth, and render a high quality hardcopy graphic of the resulting segmented area. The resources to perform this task, however, are scattered and distributed all over the network. A large *LandsatImage* of California state is stored at a mainframe called *bigblue*; the most powerful machine on the network is an UltraSparc named *speedy*, and a dedicated high-resolution graphics device is attached to a dedicated graphics processor called *picasso*. As we shall see, we rely on the ability to coordinate execution and migration of objects to complete this distributed task in an resource-efficient manner. The distributed processing of this object is illustrated in Figure 9. To actually implement such an execution, we first define the notion of a migrating object store which we call a *Execution Unit*. The *Execution Unit* holds a Private Repository to store its own collection of instance and external methods, and an instruction queue. By executing each instruction at a repository site, the *Execution Unit* is able to visit remote repositories to gather new data and methods into its Private Repository; the *Execution Unit* can thus compose its own view of an object, aggregated from distributed sources. There are a set of primitive instructions to program a *Execu-*

*tion Unit*; these are: *new*, *import*, *export*, *push*, and *invoke*.

The *new* command calls a constructor to create an object. The *import* instruction imports a repository object from a Global or Local Repository and stores the item into its Private Repository, while the *export* has the opposite effect. The *push* migrates the *Execution Unit* to a new site, akin to the *go* command in Telescript and Sumatra, where it resumes execution with the following instruction.

These instructions operate on the object in the following manner:

instruction	operation	address
import	LandsatImage	
new	LandsatImage	
import	LandsatImage__segment	
push		bigblue
import	extract	speedy
invoke	extract	
push		speedy
invoke	LandsatImage__segment	
push		picasso
import	print	picasso
invoke	print	

The client initially creates an empty *LandsatImage* object and stores it into the *Execution Unit* at *skinny*, by importing the class constructor and calling *new*. The client's goal is to segment and print a region of Santa Barbara County, unfortunately, there is no *segment* operation in the *LandsatImage* class; instead, the client must provide one. The client creates a class called *LandsatImage\_\_segment*, compiles it into byte-code, and temporarily stores it into *skinny*'s Local Repository.

```

public class LandsatImage__segment implements SSExecutable {
    public class LandsatImage__segment() {
        // constructor
    }
    public run(SSObject instance, String
        methodName, SSArglist arglist) {
        // Extract the arguments from the arglist
        // to specify a region to segment and
        // properties to segment. Then segment
        // the image stored in instance.
    }
}
  
```

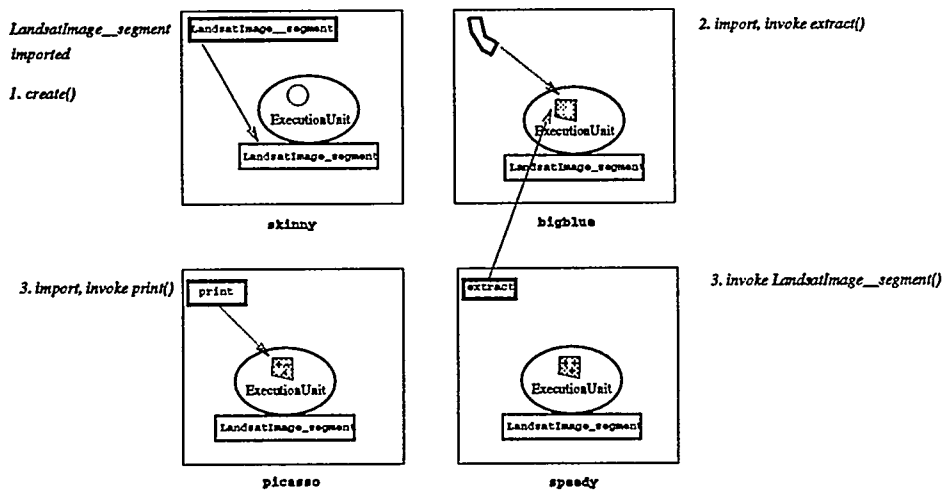


Figure 9: Remote Execution facility over 4 machines

Using the *import* command, the client loads this user-defined method into the *Execution Unit's* Private Repository for later execution at a more powerful machine.

The *push* command then migrates the *Execution Unit* to the mainframe *bigblue*, where a large *LandsatImage* of California resides. The *extract* executable, however, is stored at the UltraSparc *speedy*, leading to two possibilities: either migrate the California object from *bigblue* to *speedy* for remote execution (R3), or import the executable from *speedy* and perform the operation locally (L2). To migrate the California object, the *Execution Unit* would first have to insert the California *LandsatImage* into its repository then *push* onward to *speedy*. Unfortunately California is much too large an object to shift, consuming far too much bandwidth, the client opts for the latter alternative. The *pull* method transfers the *extract* method from *speedy's* Local Repository into the *Execution Unit's* repository, and executes it upon the California object at *bigblue*. The result of the extraction is a Santa Barbara County *LandsatImage* which is then stored in the *Execution Unit*.

Next, the *Execution Unit* proceeds to *picasso*, where it dispatches the *LandsatImage\_segment* method to mark the areas of vegetation growth. Finally the *Execution Unit* *pushes* itself to *picasso*, where it imports a specialized *print* external method from *picasso's* Local Repository, and then *imports* this *print* command to produce a hardcopy.

At this point, we briefly describe how we ensure type-safety as it appears we are permitted to dispatch any external method upon any instance, without regard to object semantics. Earlier, we noted that the *SSInstance* interface specified an invariant method, while the *SSMethod* interface required both precondition and postcondition methods. These assertion checks at run-time[Mey88] provide us with a means of checking type-safe operations. Each *StratOSphere* object specifies a class invariant, which our classloader checks whenever it dispatches a method, to ensure that the object state does not become inconsistent. Likewise, our classloader checks each *SSMethod's* preconditions and postconditions to ensure that the external method does not encounter an unexpected instance state. If any of these boolean assertions are violated, the classloader triggers an *SSException* which is returned to the client application with to report the interface assertion violation.

The *Execution Unit* provides the functionality that we require to coordinate distributed processing by incorporating both an internal private object repository and an instruction queue. The *Execution Unit* interprets the instructions at run-time; it performs these import and export instructions, by issuing *MobileObjectMessage* and *RemoteEvaluationMessages* to transfer instances and methods between repositories using COD and REV. The *StratOSphere* classloader is employed to dispatch these methods and instances to invoke a method. Similarly, the *new* instruction is processed by calling the appropriate Java object's class constructor.

The remaining *push* instruction also uses REV to transfer the *Execution Unit* to a remote site. In doing so, however, a few intricacies arise: We assume that the state of each *StratOSphere* object is safely stored in the Private Repository, so that the object can continue processing after a migration; we must also ensure that the state of the *Execution Unit* itself is maintained so that after a *push*, the subsequent instruction in the instruction queue is executed. To resolve this problem, we subclass the *Execution Unit* from the *RemoteExecutionMessage* in order to deliver it to remote repository sites for further processing, satisfying execution scenarios, L1-L4 as well as R1-R9.

## 5 Conclusion and Future Work

The final layer of interoperability in *StratOSphere* is the Distributed Applications Layer, from which distributed applications can be built with MA execution. This is the level where mobile code languages systems such as *Aglets*[LC96], *TACOMA*[JvRS95], *Telescript*[Whi96b], *Odyssey*[Whi96a], and *Voyager*[Obj96] provide services for enterprise applications. At time of writing, this final layer is still being developed and implemented for *StratOSphere*. By utilizing the services of the Distributed Object Layer, we have the migration capability of an MA system. Designing this layer, however, involves not just a mobility paradigm, but also issues of inter-agent communication, synchronization, and agent security.

Inter-agent communication varies among different agent systems. In our design, we use an idea initially developed in *MØ*[Tsc97], in which agents exchange data in a *shared memory* area of each execution site. In adapting this idea

to StratOSphere, we have reserved a portion of each Local Repository, called the *Exchange Area* to provide both data exchange and code exchange among different agents. Synchronization of updates to this Exchange Area is supported by a thread queue for each mobile process.

Work on agent security has been an area of active research[WMFS96]. Among the issues involved are scenarios in which a malicious agent may attack a host, a host may attack an agent, or an agent may attack another agent. In our design, we have greatly reduced this problem by assuming that each host repository is a trusted source; that is, only privileged administrators may maintain and update the repository. Consequently, we assume that a host will not attack an agent, but concern ourselves with the more scurrilous problem of an agent attacking the host. In our initial implementation of the Distributed Applications Layer, we use signed certificates[TV96] to authenticate each agent. No host or agent exchanges any data or information with another agent unless a signed certificate identifies the agent as a trusted source. Another approach that we are investigating includes proof-carrying code[NL96], which performs a verification of code to be executed to ensure trusted operation.

After studying all possible execution scenarios, we have designed and built a framework called StratOSphere that distributes processing across multiple host sites, and also defines a dynamic object model to implement client services. By building a multi-tiered architecture, we were able to successively develop a dynamic and extensible distributed environment for StratOSphere objects, providing REV, COD, and RCE in the Messaging layer and MA in the Distributed Object Layer. We show how the object instances and object methods can be selectively migrated to different sites for remote processing. We describe the StratOSphere naming service and object repository, and provide a programming model to build a dynamic *Execution Unit* in which sequences of method executions are targeted at remote sites for distributed processing. We have developed a prototype of the StratOSphere, and are currently investigating other applications involving MA and distributed database queries in the distributed application layer using the *Execution Unit*.

## References

- [AP95] Baruch Awerbuch and David Peleg. Online tracking of mobile users. In *Journal of the Association for Computing Machinery*, volume 42, pages 1021–1058, September 1995.
- [ARS97] Anurag Acharya, M. Ranganathan, and Joel Saltz. *Sumatra: A Language for Resource-aware Mobile Programs*, pages 111–30. Springer Verlag Lecture Notes in Computer Science, 1997.
- [BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. In *Proc. ACM Symp. on Transactions on Computer Systems*, pages 19–59, February 1984.
- [Box97] Don Box. *Creating Components with DCOM and C++*. Addison Wesley Longman, 1997.
- [Car95] Luca Cardelli. A language with distributed scope. In *Proc. of the 22nd ACM Symposium on Principles of Programming Languages*, 1995.
- [CGP<sup>+</sup>97] G. Cugola, C. Ghezzi, G. P. Picco, , and G. Vigna. A characterization of mobility and state distribution in mobile code languages. In *Proc. of the 2nd Workshop on Mobile Object Systems*, July 1997.
- [Cha97] Phil Inje Chang. *Inside the Java Web Server*. Javasoft, Inc., <http://java.sun.com/features/1997/aug/jws1.html>, 1997.
- [Che88] David R. Cheriton. The v distributed system. In *Communications of the ACM*, pages 314–33, March 1988.
- [CPV97] A. Carzaniga, G. P. Picco, , and G. Vigna. Designing distributed applications with mobile code paradigms. In *Proc. of the 19th Intl. Conf. on Software Engineering*, 1997.
- [DO87] Fred Douglass and John Ousterhout. Process migration in the sprite operating system. In *Proc. of the 7th Intl. Conf. Distributed Computer Systems*, pages 18–25, 1987.
- [Fou92] Open Software Foundation. *Introduction to OSF DCE: Rev 1.0*. Prentice Hall, Englewood Cliffs, NJ, 1992.
- [GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Sunsoft Java Series, 1996.
- [Gos91] Andrzej Goscinski. *Distributed operating systems : the logical design*. Addison-Wesley, 1991.
- [Gra96] Robert S. Gray. Agent tcl: A flexible and secure mobile-agent system. In *Proc. of the 4th Annual Tcl/Tk Workshop*, pages 9–23, 1996.
- [GRHV95] Erich Gamma, Ralph Johnson Richard Helm, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [Gun96] Shishir Gundavaram. *CGI programming on the World Wide Web*. O'Reilly & Associates, Cambridge, 1996.
- [HMPP96] J. Hartman, U. Manber, L. Peterson, and T. Proebsting. Liquid software: A new paradigm for networked systems. Technical Report Technical Report 96-11, Univ. of Arizona, June 1996.
- [Hug95] David J. Hughes. Mini sql: A lightweight database server. <http://Hughes.com.au/product/mssql>, 1995.
- [JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the emerald system. In *Proc. ACM Symp. on Transactions on Computer Systems*, pages 109–33, 1988.
- [JS96] Bill Janssen and Mike Spreitzer. *ILU 2.0 Reference Manual*. Xerox PARC, <http://ftp.parc.xerox.com/pub/ilu/ilu.html>, 1996.
- [JvRS95] D. Johansen, R. van Reneese, and F. Schneider. An introduction to the tacoma distributed system. Technical Report Technical Report 95-23, Univ. of Tromso, 1995.
- [LC96] Danny B. Lange and Daniel T. Chang. Ibm aglots workbench: Programming mobile agents in java. <http://www.trl.ibm.co.jp/aglets/whitepaper.htm>, 1996.

- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
- [NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. of the 2nd ACM Symposium on Operating System and Design and Implementation*, October 1996.
- [Obj96] ObjectSpace. Voyager technical overview. [http://www.objectspace.com/voyager/technical-white\\_papers.html](http://www.objectspace.com/voyager/technical-white_papers.html), 1996.
- [PE86] Gerald Popek and Bruce J. Walker (Ed). *The Locus Distributed System Architecture*. MIT Press, February 1986.
- [RKF92] Ward Rosenberry, David Kenney, and Gerry Fisher. *Understanding DCE*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.
- [RS91] Joel Richardson and Peter Schwarz. Aspects: Extending object to support multiple independent roles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 298–307, May 1991.
- [RWW96] R. Riggs, J. Waldo, and A. Wollrath. Pickling state in java. In *2nd Conf. on Object-Oriented Technologies and Systems (COOTS)*, pages 241–250, Toronto, Ontario, June 1996.
- [Sat96] HIRANO Satoshi. *The Magic Carpet for Network Computing: HORB Flyer's Guide*. Electrotechnical Laboratory, <http://ring.etl.go.jp/openlab/horb>, 1996.
- [SF95] T. R. Smith and J. Frew. Alexandria digital library. *Communications of the ACM*, 38(4):61–62, April 1995.
- [Sie96] Jon Siegal. *CORBA: Fundamentals and Programming*. Wiley, 1996.
- [SS97] George Semeczko and Stanley Y.W. Su. Supporting object migration in distributed systems. In *Proc. of the Fifth Intl. Conf. on Database Systems for Advanced Applications*, pages 59–66, Melbourne, Australia, April 1997.
- [Sun97] Sun Microsystems, Inc., <http://java.sun.com/products/jdk/1.1/docs/guide/reflection/index.html>. *Java Core Reflection API and Specification*, 1997.
- [Tsc97] Christian Tschudin. *The Messenger Environment MØ— A Condensed Description*, pages 149–56. Springer Verlag Lecture Notes in Computer Science, 1997.
- [TV96] Joseph Tardo and Luis Valente. Mobile agent security and telescript. In *41st IEEE Computer Society Intl. Conf.*, pages 58–63, February 1996.
- [vRHea97] Peter van Roy, Seif Haridi, and Per Brand et al. Mobile objects in distributed oz. In *Proc. ACM Symp. on Transactions on Programming Languages and Systems*, pages 804–51, September 1997.
- [WAAS97] D. Wu, D. Agrawal, A. El Abbadi, and A. Singh. A java-based framework for processing distributed objects. In *Proc. Intl. Conf. on Conceptual Modeling*, pages 333–46, Los Angeles, CA, 1997.
- [Whi96a] James White. Mobile agents white paper. <http://genmagic.com/agents/Whitepaper/whitepaper.html>, 1996.
- [Whi96b] James White. Telescript technology: Mobile agents. <http://genmagic.com/TeleScript/WhitePapers>, 1996.
- [WMFS96] J. D. Guttman W. M. Farmer and V. Swarup. Security for mobile agents: authentication and state appraisal. In *4th European Symposium on Research in Computer Security Proceedings*, pages 118–30, September 1996.
- [WRW96] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for java. In *2nd Conf. on Object-Oriented Technologies and Systems (COOTS)*, pages 219–231, Toronto, Ontario, June 1996.