MAGE: A Distributed Programming Model

By

Earl T. Barr

B.A. (University of the Pacific, Stockton) 1988
M.S. (University of California, Davis) 1999

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Raju Pandey (Chair)

Professor Premkumar Devanbu

Professor Ronald Olsson

Committee in Charge

2009

Earl T. Barr
June 2009
Computer Science

MAGE: A Distributed Programming Model

Abstract

This thesis explores computation mobility. We view mobility, the selection of an execution environment, as an attribute of a computation. To capture this attribute, we introduce a novel programming abstraction, which we call a *mobility attribute*, that specifies where computations should occur within a distributed system. Programmers dynamically bind mobility attributes to program components, nonempty sets of functions and their state. Once bound to a component, mobility attributes apply prior to each execution of that component. For example, a mobility attribute can specify that a computation should be collocated with an invoker. When a component bound to that mobility attribute receives an invocation it moves to the invoker's location before executing.

Mobility attributes are the primitives that form MAGE, a new programming model that allows a programmer runtime control over the location of a program's components. This control can improve the program's robustness or its runtime efficiency by co-locating components and resources. To illustrate the utility of MAGE, this thesis presents the design and implementation of the MAGE programming model in a Java library that extends Java's RMI to support mobility attributes.

To my family — Kerrean, Nathan, and Nicolas

Contents

List of Algorithms

List of Figures

List of Listings

List of Tables

Acknowledgments

I could not have accomplished this task alone.

I would like to thank my advisor, Raju Pandey, for his patience and for the time he devoted to me even while on leave to found and run his startup.

Ron Olsson and Prem Devanbu served on both my dissertation and qualifying exam committees. I am grateful to Ron for his time, and his prompt and careful feedback. I am grateful to Prem for showing that there is room in research for polymaths. I am grateful to Felix Wu who has been unstinting with his time and unfailingly positive. I am especially grateful to Zhendong Su, without whose generous support, both intellectual and financial, I would not have been able to finish this dissertation.

I want to thank Kim Reinking for her belief in me when my drive to finish flagged.

Talking and working with my fellow graduates students has been the best part of my graduate school experience. Most of the computer science I know, I learned from them. I have been fortunate to work with and learn from two generations of graduate students.

The first generation includes Michael Haungs, to whose mad skillz I pay homage, Eric Hyatt, who has been my omnipresent sounding board, Aaron Keen, Fritz Barnes, Ghassan Misherghi, and Scott Malabarba.

The second includes Dimitri DeFiguieredo, whose intellectual excitement is contagious, David Hamilton, who challenges me to be precise, Mark Gabel, Rennie Archibald, and Christian Bird.

I want to thank Michael Nadler who helped me design and implement MAGE.

I am grateful to my wife Kerrean, whose understanding, encouragement, and patience made this dissertation possible. This accomplishment is as much hers as mine.

Chapter 1

Introduction

Mobility is the true test of a supply system.

Captain Sir Basil Liddell Hart
Thoughts on War, 1944

The days of single address space computing are behind us. Driven by the power wall [77], multicores are upon us — dual core machines are already ubiquitous, 16 core machines are commercially available [35], and Intel Corporation is prototyping 80 core machines [65]. To increase server utilization, hosting services are employing virtualization to partition the resources of a single machine, turning it into a virtual cluster [62]. The World Wide Web (WWW) [11] and peer to peer applications, like BitTorrent [84] and Gnutella [34], are inherently distributed. Scientific computation is moving from its traditional supercomputer environment to distributed systems, lured by the expandability and cost savings. To capitalize on this trend, companies now rent out processor farms [104, 114]; in academia, the Grid [20] has similar goals.

In short, a wide variety of services and data is dispersed on architectures that are heterogeneous and evolving. To cope with such change, distributed systems must support resource discovery, incorporate new hardware, and handle variation in resource availability such as network latency and bandwidth. For example, over time a host that was CPU-bound may become idle, and one data source may be exhausted while another comes online.

To fully exploit the runtime environment, distributed programming models should provide

mechanisms that allow programs to control where their computations occur and thereby support load balancing, respond to network congestion, and adapt to the appearance, disappearance, and shifting of resources. This thesis presents one such programming model.

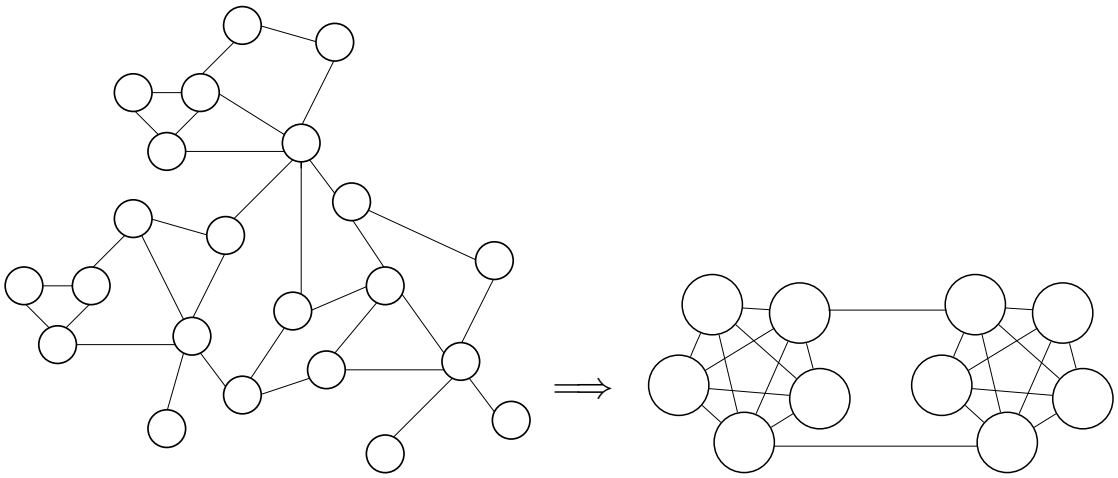
1.1 The Layout Problem

By definition, a distributed application spans machines and comprises multiple, intercommunicating components. A component has state and code, and communicates with other components via messages. For example, the application could be a Web 2.0 service [115] that runs across a browser on a home PC, middleware servers, and a backend database. At a coarse granularity, the browser, the middleware, and the backend database are components; at a finer-grain, components may be functions or closures or objects, depending on the implementation language. This thesis focuses on this latter, finer-grained view of components.

Figure 1.1a captures a distributed application's component interdependency graph, the components that may intercommunicate in the lifetime of the application. The vertices are components. The edges model communication channels. The principle of locality [24] means that, at any point in time, the active components in an application's working set comprise a subgraph of Figure 1.1a.

Figure 1.1b represents that application's execution environment, which could be a symmetric multiprocessor, a cluster, or a home PC interacting with servers. The nodes in Figure 1.1b are execution engines and are not created equal. Some possess unique resources, like a user or a database. The edges subsume the interconnection tissue of the network, such as routers and switches.

Controlling where a distributed application's computations occur boils down to layout — the problem of mapping that application's component graph onto a network of execution engines. All distributed applications must confront the layout problem, which Figure 1.1 depicts. Both graphs are dynamic — the application can create and destroy components, while execution engines can come online or go offline and their resource profiles can change. Thus, an application must be able to dynamically adapt and refine its layout as it learns



(a) Component Intercommunication (b) Execution Environment

Figure 1.1: The Layout Problem

more about its environment.

A *migration policy* is a particular layout function. It specifies when and where a component should execute, and thus move. Two axes span the space of migration policies — the "static versus run-time" and "automatic versus manual" continua.

Optimal automatic layout is intractable [33, Graph Partitioning], so migration policies are often either manual or heuristic. Deployment specifications are migration policies that statically map components to execution engines, where they remain for the life of the application. Such specifications are usually manual, but tools exist that tackle static automatic layout, notably Coign [46] and J-Orchestra [100]. These tools employ heuristics, simplify the problem, and resort to human aid. Gang scheduling forms policies that automatically change the layout of running applications at regular intervals [75]. FarGo allows administrators to manually reconfigure an application's layout at runtime [42].

Programmers can use programming models that control component location to write migration policies that change an application's layout at runtime [63, Chapter 3]. These policies allow the application to adapt to its environment. They straddle the automatic versus manual continuum: it is special-purpose code that a programmer tailors for a particular application and environment.

This thesis falls into this last category. It presents a novel programming model for writing migration policies that apply at runtime.

1.2 Mobility

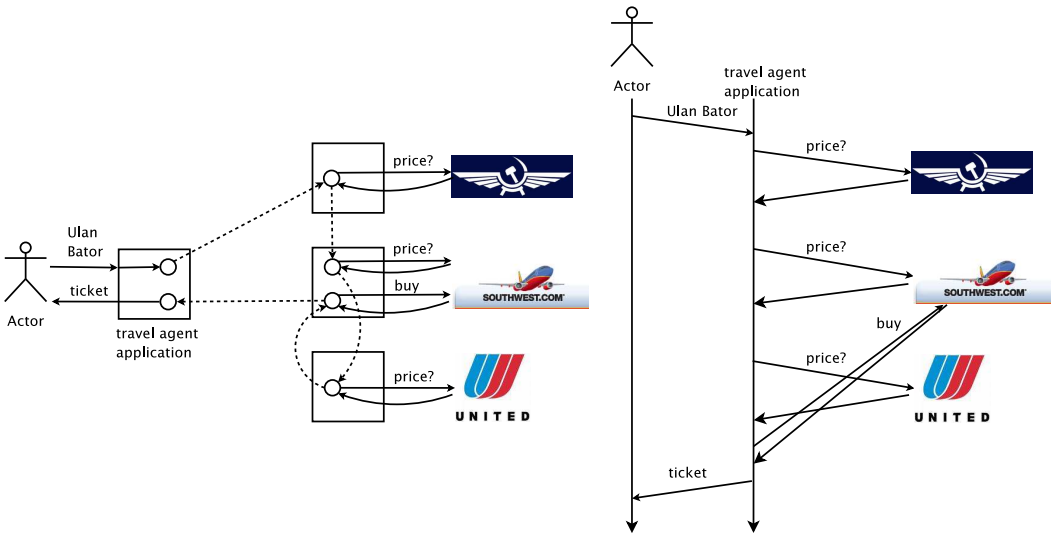
A travel agent application is often used to motivate mobile code, *e.g.*, [109]. Figure 1.2 depicts two different implementations. Figure 1.2a uses mobile code, while Figure 1.2b uses messages.

In Figure 1.2a, the application creates mobile code that travels the network searching for the lowest fare to Ulan Bator, Mongolia. First, it searches all sites. Then it returns to the site with the lowest fare, Southwest Airlines, where it purchases a ticket. Finally, it returns to its origin and presents the user with a ticket.

This movement of code from one site to another, denoted by dashed lines in Figure 1.2a, is just a message that contains the mobile code. In addition to the logic it uses to determine which ticket to buy, this code must specify its itinerary, as well as contain the messages and protocol logic for interacting with the airlines. The benefit of mobile code is that it converts the messages that interact directly with the airlines from remote to local messages. The performance gain of this conversion depends on the cost of migration.

Whatever the initial cost of migration, that cost increases as the code gathers state. In Figure 1.2a, the mobile code collects and carries with it the lowest price it has yet seen. When the actor and the airlines are in different administrative domains, two security concerns arise: first, servers must accept and *execute* untrusted code; and second, clients must trust that no server alters their code or its data. All attempts to address these security concerns either add messages or additional state, such as proofs, to authenticate and validate mobile code and its payload. These efforts worsen performance.

In contrast, Figure 1.2b's message-based implementation sends only data in its messages. Thus, all of its messages and their replies are remote messages that must traverse the network. The size of these messages are independent of the number of ticket vendor sites searched. Because intermediate results return to the actor's machine, the actor can monitor



(a) Mobile Code (b) Message Passing

Figure 1.2: The Travel Agent Application

the progress of the search and potentially modify it. The message passing model only forces airlines to expose a narrow API that supports the relevant ticket search protocol, not an execution engine, which exposes a much larger surface for an attacker to probe.

Although a changing environment demands runtime approaches, static manual deployment has dominated server applications, ironically even this travel agent application, the canonical example of a mobile agent application. For years, mobile programming languages have not taken off outside of research prototypes. Like functional languages, mobile code has had ardent supporters, but little traction in industry. The reason is simple. For many applications, like the travel agent example, a message-passing implementation requires less bandwidth and raises no security concerns related to executing untrusted code. Additionally, the rate of change in computing environments has been slow enough for static approaches to be adequate. Put another way, mobile code has lacked a "killer" application.

In spite of its problems, mobile code has always found its niches, especially within a single administrative domain. Some applications use protocols, subprotocols of which can be delegated to a component. Such delegation is profitable when the savings from collocating the participants in that subprotocol exceeds the cost of migration. Applications

for which subprotocol delegation pays for itself form one such niche. The travel agent application as presented does not fall into this niche because the site-specific subprotocol, shown in Figure 1.2a, consists of a single message exchange. If instead, the mobile component exchanged many messages with each airline it visited, the savings of collocation might have exceeded the cost of migration.

Another niche is formed by applications that interact with large datasets and do not know in advance how to group the elements in those datasets into result sets. Such an application could build its result set, element by element, by sending a message to request each element, at great cost. Of course, a single, specialized message that defines a specific result set would be more efficient, but we do not know that results set in advance. In these situations, a program is the most flexible and compact way to define a result set. When sent as messages, these programs are mobile code. Complex SQL queries dispatched to remote DBMS are one example of such programs. If the SQL queries a client could send were replaced with distinct messages, their number would be bounded only by the combinatorics of a database's attributes and values.

Today, extensible browsers have changed the economics of startups [40]. Entirely new services, like Gmail, run user interfaces and perform calculations, in their clients' browsers. The browser has evolved from a markup renderer to an extensible execution engine that runs code sent by servers. The Web 2.0, built on the WWW using javascript and flash, is mobile code's killer application.

Web 2.0 applications are easy to deploy and cheap to maintain. Unlike standalone binaries, both initial installation and updates can happen whenever a user accesses the application. These applications demonstrate that mobility can improve performance when migration is cheap. Web 2.0 applications use mobile code principally in their user interfaces to eliminate messages and network latency. Google Docs could not work without it.

Web 2.0 applications fundamentally rely on trust, although they do utilize techniques such as sandboxing, certificates, and encryption. If you visit a Web 2.0 service, you trust content *including code* from that service. Like Web 2.0, this thesis presents a programming model that trusts both the components and the execution engines those components run on.

1.3 MAGE

When invoked, a mobile component can move before executing. In Listing 1.1, c is a mobile component and f is an operation on c. The operation f may need resources external to c. The invoker i runs the invocation in Listing 1.1. Relative to the invoker i, c is either local or remote. In a dynamic, evolving distributed system, a program may wish to collocate c and the external resources it needs or collocate c and i prior to c's execution. This placement problem becomes even more complicated when c needs more than one resource from different hosts.

Listing 1.1: An Invocation

$x = c.f(p_1, p_2)$;

Code requires a processor and storage to execute. We contend that mobile code requires a third attribute, a migration policy, to address the layout problem. We present a novel programming abstraction, called a *mobility attribute*, that encapsulates a migration policy and binds to a program's components. Before invoking c, the invoker i binds a mobility attribute m to c. When i invokes c in Listing 1.1, m selects an execution engine to which c moves and executes.

A mobility attribute is the migration policy for the component to which it is bound. A program is an aggregation of components. The mobility attributes that a program binds to its components defines the migration policy of that program as a whole.

Mobility attributes capture all programming models proposed to date that incorporate dynamic layout. They isolate migration logic from an application's core logic. Because a mobility attribute only moves a component when that component receives an invocation, no time is wasted moving components not in use. Using mobility attributes, distributed applications can employ migration policies that move components at runtime. They can also bind different attributes to a component, as the runtime environment evolves. Finally, programmers can combine simple mobility attributes to form attributes that apply complex policies. For example, a programmer could combine an "execute on the least loaded host"

attribute with one whose policy is "execute on the host with the most available bandwidth" to form a new attribute whose policy is "execute on the least loaded host while compute bound, then move to the host with the most available bandwidth to transmit results."

This thesis presents and investigates the properties of MAGE (Mobility Attributes Guide Execution), a distributed programming model based on mobility attributes. MAGE

- adapts the layout of a program's components, at runtime, to the underlying computation and communication infrastructure;
- encapsulates all current programming models that incorporate dynamic layout;
- separates an application's core logic from the exigencies of dynamic layout;
- moves only those components in an application's working set; and
- realizes migration policies as first class objects that can be composed.

The principal contribution of this thesis is the MAGE programming model and its realization. The novelty of MAGE rests on the above features it combines. MAGE defines distributed invocations as configurations of invoker and invoked. The analysis of all combinations of these configurations leads to the discovery of heretofore neglected configurations and allows MAGE to encompass all proposed mobile programming models. Isolating an object migration policy into mobility attributes relieves the programmer of the burden of simultaneously thinking about layout and application logic. MAGE integrates invocation and mobility, so it applies a migration policy only to objects currently exchanging messages, *i.e.* currently in the working set. This relieves the programmer of the difficult task of, and the performance cost of incorrectly, statically inferring such working sets. MAGE's attribute composition can construct complex policies from simple policies. MAGE provides these qualitative benefits at low cost (Section 7.1). A programmer would choose MAGE for projects that require dynamic layout adaptation because MAGE (1) separates the migration concern into attributes; (2) facilitates policy reuse via attribute composition; and (3) offers powerful, flexible, and elegant control over object and invocation placement.

MAGE is best suited for applications whose optimal component placement is highly uncertain statically, because of variation in their execution environment. Load-balancing is a case in point. A MAGE application can use mobility attributes to push the load-balancing decision into clients, thereby eliminating a dedicated load-balancing tier and improving scalability. For example, mobility attributes could round-robin SQL queries across a collection of database servers.

The rest of this thesis is organized as follows. Chapter 2 begins with a gentle introduction to MAGE in which we re-implement Sun's RMI tutorial under MAGE and contrast the two solutions. Chapter 3 presents the semantics of MAGE's mobility attributes. Chapter 4 presents the MAGE programming model, its primitives and operators. We show how mobility attributes arose from an analysis of existing distributed programming models that incorporate mobility, discuss the MAGE work published at ICDCS 2001 [8], and then generalize to increase the power and flexibility of mobility attributes. Chapter 5 compares forwarding pointers against a centralized directory, and shows that the invocation protocol MAGE initially used, the self-routing invocations protocol, which combines lookup and invocation messages into a single message, uses more bandwidth on average than the "find, then invoke" protocol, which uses two distinct messages. Chapter 6 describes the implementation of RMI, then presents the implementation of MAGE as a set of extensions to RMI. Chapter 7 measures the overhead of our implementation of mobility attributes and presents peripatetic dining philosophers, a variant of dining philosophers with mobile philosophers. Chapter 8 places MAGE into the context of its related work. Chapter 9 summarizes the thesis and outlines future work.

Chapter 2

Two Π Tutorials

Example is always more efficacious than precept.

Samuel Johnson

Rasselas, 1759

In July 2007, Sun's remote method invocation (RMI) tutorial presents a generic framework for moving code from a client to server for execution [69]. The code the client sends to the server calculates π. Here, we introduce RMI and describe Sun's tutorial. We then describe its re-implementation under MAGE and compare the two implementations.

2.1 RMI

Java's Remote Method Invocation (RMI) mechanism allows an object, the client, running in one Java virtual machine to invoke methods on an object, the server, running in another Java virtual machine. These invocations are the arcs labeled RMI in Figure 2.1.

In Figure 2.1, the server object first registers with the `rmiregistry`, as shown by the RMI arc from the RMI server to the `rmiregistry`. Then the client downloads a stub, or proxy, for that server from the `rmiregistry`. For this to work, both client and server must statically share the name the server used when it registered in the `rmiregistry`.

To send an invocation, the client locally invokes a method on the stub. The stub

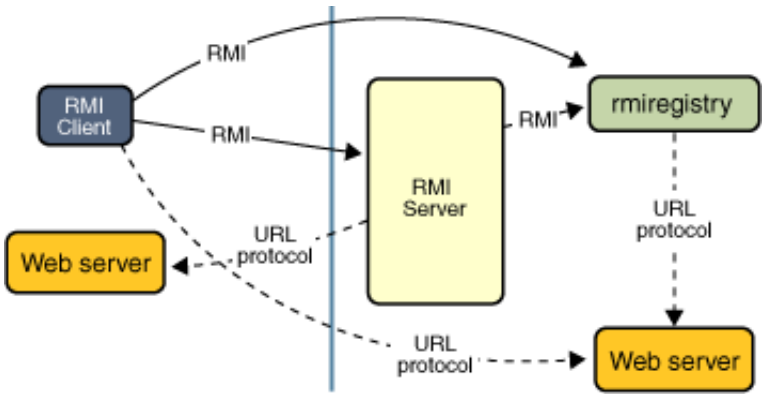


Figure 2.1: RMI Overview

marshals that invocation and sends it to the server. The arc directly from `RMI Client` to `RMI Server` represents such an invocation. Upon receiving the invocation, the server unmarshals the invocation, decodes the method, executes that method in its address space, and replies with the result.

Listing 2.1: The Compute Interface

```
1 package compute;
2
3 import java.rmi.Remote;
4 import java.rmi.RemoteException;
5
6 public interface Compute extends Remote {
7     <T> T executeTask(Task<T> t) throws RemoteException;
8 }
```

For this to work, both the client and the server must share this method's signature. Here, that shared interface is `Compute`, shown in Listing 2.1. `Compute` defines the `executeTask` method. Figure 2.2 illustrates the `executeTask` protocol.

The `executeTask` method contains a formal of type `Task`. Listing 2.2 defines `Task`. As noted, Sun's RMI tutorial presents a framework that allows clients to send code to servers. This technique is called remote evaluation (REV) [93] and is discussed in greater depth in

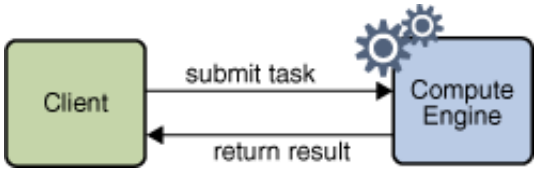


Figure 2.2: Compute Task Protocol

Chapter 3. It is this `Task` interface that implements REV in this example.

Listing 2.2: The Task Interface

```
1 package compute;
2
3 public interface Task<T> {
4     T execute();
5 }
```

RMI supports the code mobility of parameters, other than a method's receiver, or **`this`**, parameter. When a client wants to pass subclasses or classes that implement an interface (the case here) as actuals to a RMI invocation, the classes of the actuals may be unknown to the server. RMI annotates each invocation with a `codebase` attribute, which contains a URL that points to the client's codebase that contains such classes. When the server cannot resolve a class, it attempts to find the class at that URL using its `URLClassLoader`.

Thus, `Task` allows clients to define classes that are statically unknown to the server, but that the client can send for execution on the server.

In Figure 2.1, the web servers serve these codebases. When the server object registers itself, the `rmiregistry` downloads the server object's class from the web server running on the same host as the server object. When the client sends unknown classes that implement `Task` to the server, the server downloads those classes from the client-side web server.

2.1.1 Server

Listing 2.3 depicts the initialization of the `ComputeEngine` server. To dynamically exchange a stub, an RMI client and server must statically share a name under which to upload and

download that stub in the rmiregistry. Line 25 contains the declaration and initialization of such a name.

In lines 27-8, the server implicitly starts the RMI infrastructure, publishes `engine`, and receives a stub for engine. At this point, if the client had a statically deployed stub or the server could send `stub` to the client, the client could invoke `executeTask` on `engine`. In our example, the client dynamically acquires its stub, so the server gets a stub for the `rmiregistry` on line 29, and then binds the name `Compute` to `stub` in the `rmiregistry` on line 30.

2.1.2 Client

In Listing 2.4 on line 14, the client initializes the name that it statically shares with the server in order to dynamically exchange stubs via the `rmiregistry`. On line 15, the client gets a stub to the `rmiregistry` running on `arg[0]`, the server's host. The client immediately, line 16, uses that stub to get a stub to the server's `ComputeEngine` instance, `engine`.

On line 18, the client instantiates the `Pi` class. Its class definition follows.

```
public class Pi implements Task<BigDecimal>, Serializable
```

`Pi` must implement the `Task` interface so that the server can execute it; it must implement the `Serializable` interface so that it can be marshaled. `Pi` implements Machin's formula [105], Equation 2.1, to compute π. We do not show the rest of `Pi`'s definition because its details are incidental.

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239} \tag{2.1}$$

To start the server, we first start the `rmiregistry`, then the `ComputeEngine`, as shown in Listing 2.5.

Listing 2.6 depicts running `ComputePi` on a client. `ClassFileServer` is a minimal web server that accepts requests for Java classes. It fills the role of the web server in

Listing 2.3: ComputeEngine

```
1  package engine;
2
3  import java.rmi.registry.LocateRegistry;
4  import java.rmi.registry.Registry;
5  import java.rmi.server.UnicastRemoteObject;
6  import compute.Compute;
7  import compute.Task;
8
9  public class ComputeEngine implements Compute {
10
11    public ComputeEngine() {
12       super();
13    }
14
15    public <T> T executeTask(Task<T> t) {
16       return t.execute();
17    }
18
19    public static void main(String[] args) {
20       if (System.getSecurityManager() == null) {
21          System.setSecurityManager(new SecurityManager());
22       }
23       try {
24          String name = "Compute";
25          Compute engine = new ComputeEngine();
26          Compute stub = (Compute)
27             UnicastRemoteObject.exportObject(engine, 0);
28          Registry registry = LocateRegistry.getRegistry();
29          registry.rebind(name, stub);
30          System.out.println("ComputeEngine bound");
31       } catch (Exception e) {
32          System.err.println("ComputeEngine exception:");
33          e.printStackTrace();
34       }
35    }
36 }
```

Listing 2.4: `ComputePi`

```
1  package client;
2
3  import java.rmi.registry.LocateRegistry;
4  import java.rmi.registry.Registry;
5  import java.math.BigDecimal;
6  import compute.Compute;
7
8  public class ComputePi {
9     public static void main(String args[]) {
10       if (System.getSecurityManager() == null) {
11          System.setSecurityManager(new SecurityManager());
12       }
13       try {
14          String name = "Compute";
15          Registry registry =
16             LocateRegistry.getRegistry(args[0]);
17          Compute comp = (Compute) registry.lookup(name);
18          Pi task = new Pi(Integer.parseInt(args[1]));
19          BigDecimal pi = comp.executeTask(task);
20          System.out.println(pi);
21       } catch (Exception e) {
22          System.err.println("ComputePi exception:");
23          e.printStackTrace();
24       }
25    }
26 }
```

Figure 2.1. It listens at port 2001, which `codebase` reflects. This RMI implementation requires `ClassFileServer` so that `engine` server object can request and download the `Pi` application.

2.2 MAGE

MAGE provides programmers with the class `MageMobileObject`. When an instance of `MageMobileObject` is invoked, all of an invocation's parameters, in particular the instance itself (the receiver of the invocation), are mobile. This means an application can deploy classes, and their code, simply by instantiating instances of `MageMobileObject`

Listing 2.5: Running `ComputeEngine`

```
[barr@ido apps]$ ./rmi-pi-run -s
Starting rmiregistry
rmiregistry&
Starting ComputeEngine server
(java -jar rmi-pi-server.jar)
ComputeEngine bound
```

Listing 2.6: Running RMI `ComputePi`

```
[barr@segou rmi]$ run -j . -c ido 100
Starting ClassFileServer
(java -jar ./rmi-pi-classserver.jar&)
Starting Pi client
(java -Djava.rmi.server.codebase=http://segou.cs.ucdavis.edu:2001 \
//home/barr/work/mage/dev/trunk/app/pi/rmi/classes/ -jar \
./rmi-pi-client.jar ido 100)
3.14159265358979323846264338327950288419716939937510582097 4 \
94459230781640628620899862803482534211706 80
```

and invoking their operations.

Listing 2.7 illustrates how MAGE simplifies the server-side of the RMI tutorial. MAGE transparently manages mobile object stubs, so there is no need to explicitly exchange them in the application code. Further, the client and server have no need to statically share the name of a stub. A MAGE server simply executes the methods of mobile objects that visits it, so MAGE dispenses with RMI's shared interfaces, such as `Compute`.

The only programmer-visible action here is to initialize the MAGE infrastructure on line 13. The corresponding activity in RMI occurs implicitly when the server publishes `engine` via `UnicastRemoteObject.exportObject`, on lines 26-7 of Listing 2.3.

MAGE directly supports mobility. Thus, there is no server object, like `engine` in the RMI solution. Nor is there any need of the cumbersome indirection of the `Task` interface to move the `Pi` class to the server. Instead, `Pi` extends `MageMobileObject` and moves to the server, as we see in Listing 2.8.

MAGE's version of `ComputePi` dispenses with the shared name and any interactions with the `rmiregistry`. It adds the declaration and binding of an REV mobility attribute,

Listing 2.7: MAGE Server Initialization

```
1  package server;
2
3  import java.rmi.RemoteException;
4  import mage.MageServer;
5
6  public class StartMage {
7
8     public static void main(String[] args) {
9        if (System.getSecurityManager() == null) {
10          System.setSecurityManager(new SecurityManager());
11       }
12       try {
13          MageServer.init();
14          System.out.println("Mage Initialized");
15       } catch (Exception e) {
16          System.err.println("Unable to initialize Mage:");
17          e.printStackTrace();
18       }
19    }
20 }
```

on lines 11 and 13. Chapter 3 is dedicated to mobility attributes. For our immediate purposes, a mobility attribute moves an object when that object is invoked. Here, the mobility attribute `rev` takes the server's name `arg[0]` as its target. When the `Pi` instance `task` is invoked on line 14, `rev` moves it to the server where it executes. After printing out the result, `main` calls `System.exit(0)` because instantiating the `task` object implicitly calls `MageServer.init()` because all MAGE hosts are potentially servers. The `exit` call simply mirrors the behavior of the RMI version.

Listing 2.8: ComputePi in MAGE

```
1  package client;
2
3  import java.math.BigDecimal;
4
5  public class ComputePi {
6     public static void main(String args[]) {
7        if (System.getSecurityManager() == null) {
8           System.setSecurityManager(new SecurityManager());
9        }
10       try {
11          REV rev = new REV(args[0]);
12          Pi task = new Pi(Integer.parseInt(args[1]));
13          task.bind(rev);
14          BigDecimal pi = task.execute();
15          System.out.println(pi);
16          System.exit(0);
17       } catch (Exception e) {
18          System.err.println("ComputePi exception:");
19          e.printStackTrace();
20       }
21    }
22 }
```

Listing 2.9: Running a MAGE Server

```
[barr@ido apps]$ ../dev/trunk/bin/mageregistry
Starting mageregistry
[barr@ido apps]$ java -Dmage.class.path=‘pwd‘/rmi-pi-compute.jar \
-jar mage-pi-server.jar
Mage Initalized.
```

Under MAGE, only one line in the class `Pi` changes:

```
public class Pi implements Task<BigDecimal>, Serializable
```

becomes

```
public class Pi extends MageMobileObject
```

Although the MAGE server that handles incoming `HTTP` requests for classes is not programmer visible, it is still present. Under MAGE, class serving functionality is built-in;

Listing 2.10: Running MAGE `ComputePi`

```
[barr@segou mage]$ java -Dmage.class.path= \
/home/barr/work/mage/dev/trunk/app/pi/mage/mage-pi-client.jar \
-Djava.security.policy=client.policy client.ComputePi ido 100
3.14159265358979323846264338327950288419716939937510582097 4 \
94459230781640628620899862803482534211706 80
```

under RMI, such functionality is application-specific. The RMI implementation provides it via `classserver`, which must be explicitly started. The `mage.class.path` property sets the path where the MAGE class server looks for classes. The RMI implementation pulls this setting from the codebase property in Listing 2.6.

For comparison with Listings 2.5 and 2.6, Listings 2.9 and 2.10 contain transcripts of running the MAGE version of `Pi`.

2.3 Summary

In this chapter, we have presented Sun's RMI tutorial, which uses a statically shared interface and parameter mobility to dispatch a π calculator to a server.

We then re-implemented this application in MAGE more simply. We eliminated the need for a shared interface, as well as the server-side `engine` object. We made the `Pi` class mobile, instantiated it, and bound that instance to a remote evaluation mobility attribute. When we invoked that instance, the mobility attribute dispatched the `Pi` instance to its remote target for execution. In so doing, we have used mobility attributes without defining them, or the programming model built on top of them.

Chapter 3

Mobility Attributes

While high over your heads you will see the best best
Of the world's finest, fanciest Breezy Trapeezing
My Zoom-a-Zoop Troupe from West Upper Ben-Deezing
Who never quite know, while they zoop and they zoom,
Whether which will catch what one, or who will catch whom
Or if who will catch which by the what and just where,
Or just when and just how in which part of the air!

Dr. Seuss
If I Ran The Circus, 1956

In this chapter, we introduce mobility attributes and define their semantics. We rely on set theory and operational semantics. Our goal is to capture the essence of MAGE using a language independent notation to emphasize MAGE's universality and applicability to a wide variety of languages.

In Section 3.1, we define the terminology used in this chapter. In particular, we define execution engine and component.

Section 3.2 analyzes existing programming paradigms in which invocation triggers component motion. Mobility attributes arose from the realization that these paradigms can

be characterized by the locations of an invoker and the invoked program component before and after the invocation or as a pair of hosts — a starting host and a target host.

In Section 3.3, we define primitive mobility attributes to be precisely these pairs, then show how existing programming paradigms fit into a taxonomy formed from these mobility attributes. Classifying these paradigms yields a new paradigm, and surprisingly reclassifies one paradigm as an instance of another. We then show how a programmer can bind these attributes to a program's components to control a program's layout. These primitive mobility attributes and their related programming model, form the work we presented at ICDCS in 2001 [8].

In Section 3.4, we lift the start and target hosts of primitive mobility attributes to sets of start and target hosts. This generalization is natural and allows us to relax primitive mobility attributes to cover a wide variety of configurations. In particular, it eliminates the need to coerce a primitive mobility attribute when we do not care where a component starts, *i.e.* where it receives an invocation, but only that it execute on a specified target.

Dynamic attributes allow programs to use mobility attributes to define migration policies that react and adapt to a program's environment at runtime. When policies, such as "execute on lightly loaded systems" and "execute on systems that have a resident DBMS," complement each other, MAGE allows a programmer to compose them. Section 3.5 presents mobility attribute composition and shows how a programmer might use composition to create complex migration policies using simpler attributes as building blocks.

All the attributes presented to this point are client-side attributes. In Section 3.6, we introduce server-side mobility attributes that bind directly to a component. These attributes allow a component, in its role as a server, input into where it executes. For example, programmers may wish, due to security concerns, to restrict where a component moves and executes. Server-side mobility attributes allow them to express such policies.

3.1 Terminology

Definition 3.1.1 (Execution Engine)**.** An *execution engine* or *host* runs code. Execution engines do not nest. They define a scope in which resources exist.

Remark. Because "execution engine" is a cumbersome phrase, we variously use host and location as synonyms for it in the text that follows wherever that substitution is clear from context. An execution engine may be realized as a virtual or physical machine, or as a single system image (SSI) cluster. The fact that execution engines do not nest implies that they do not move.

Definition 3.1.2 (Component)**.** A *component* is a nonempty set of functions, and their associated data state. Components do not nest. Components reside within an execution engine. Components can *move*, or change execution engines. An *active* component has execution state — stack and register file. An *inactive* component has no execution state.

Remark. An active component is an *actor* [41]; it has its own thread of control. Since an inactive component is just an active component with no execution state, we assume active components below, unless otherwise noted.

We have kept these definitions simple to closely reflect existing practice. Next, we use them to model existing distributed programming paradigms, as well as propose new ones.

3.2 Integrating Invocation and Mobility

Listing 3.1, which repeats Listing 1.1 in Chapter 1, is an invocation statement in a component of a distributed program. As before, c is a mobile component; i is the invoker. This invocation statement differs from a local procedure call (LPC), in that i and c are not necessarily collocated. What must happen when i calls an operation on c? Since c could be at a different host than i, the system must find it. Let H denote the set of all hosts that comprise a MAGE system. Let C be the set of all components in a distributed program, so $c, i \in C$[1].

[1] C is the vertex set of Figure 1.1a; H is the vertex set of Figure 1.1b.

When i and c are not collocated, i needs some way to identify c. Let I be the set of unique identifiers for C. If c is at the remote host $r \in H$, then the system must marshal the call's arguments[2], forward the arguments to the component, execute the call at r, and return the result to the invoker i.

Listing 3.1: An Invocation

$x = c.f(p_1, p_2)$;

When the location of the remote component is fixed and statically known, the find is superfluous and we have a *remote procedure call* (RPC), as defined by Birrell *et al.* [13]. Figure 3.1 depicts an RPC invocation. Let find : $I \rightarrow H$ be the function that returns a component's location. When find$(id_i) = l$ is the local host of i, then $R = H - \{l\}$ is the set of all hosts remote to l. From l, the invoker i calls c, which runs on $r \in R$. While running, c accesses the resource d. When c finishes the computation, it returns the result to i. A *remote method invocation* (RMI) is an object-oriented form of RPC in which the remote object is an implicit parameter of the procedure call. Although Birrell *et al.*'s original definition of RPC and Java's RMI differ in some technical details, we give pride of place to RPC since it came first and consider it the superset and RMI the subset[3]. Further, we use RMI to denote any implementation of RPC in an object-oriented language, and consider Java's RMI an implementation of RMI in Java.

Now consider adding component mobility to the programming model. Find is no longer static. Component location becomes a primitive and requires operators to manipulate it. These operators can either be explicit or implicit. An obvious explicit operator is move(c, t), which moves the component c to the target $t \in H$.

An elegant, implicit operator on component location integrates mobility and invocation: receiving an invocation triggers code motion. Component invocation is a natural time

[2]Marshalling is the process of translating an invocation's parameter types and actuals into a format suitable for network transfer.

[3]Birrell *et al.*'s definition of RPC identified methods by both name and version to ease the deployment of new behavior. Java's RMI does not provide this functionality. On the other hand, as discussed in Chapter 2, an invocation in an object-oriented language can contain actuals that are subclasses of a formal in a method's signature. If such a subclass is unknown to a component when that component receives the invocation, RMI allows the component to discover and download the subclass, and its function definitions.

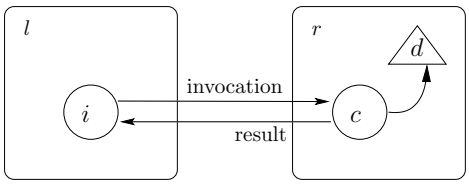


Figure 3.1: Remote Procedure Call

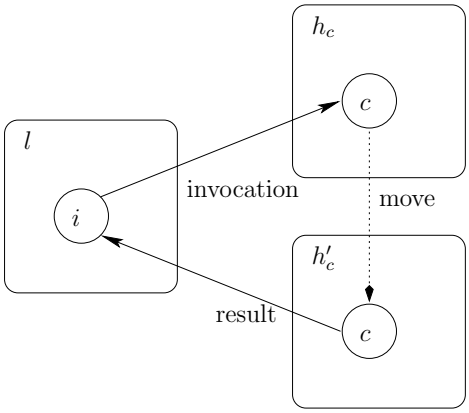
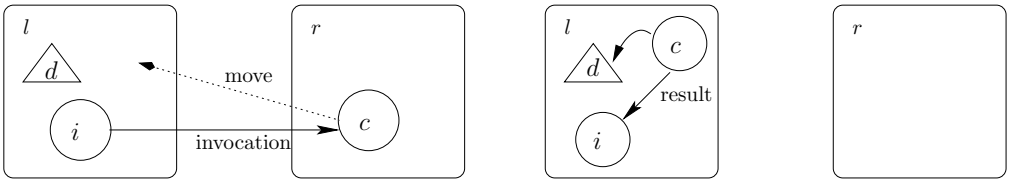


Figure 3.2: Mobile Invocation

to move a component for two reasons: first, the principle of locality assures us that the component is in use and that the work required to move it is less likely to be wasted and, second, at that time we can best decide where to move the component, since an application can use its state at the time of the call as well as the current state of the system. In short, the location of a component is of no import until the component is needed. Further, integrating mobility with invocation eliminates the clutter of explicit move calls in an application's source code: it separates an application's core logic from its mobility aspect.

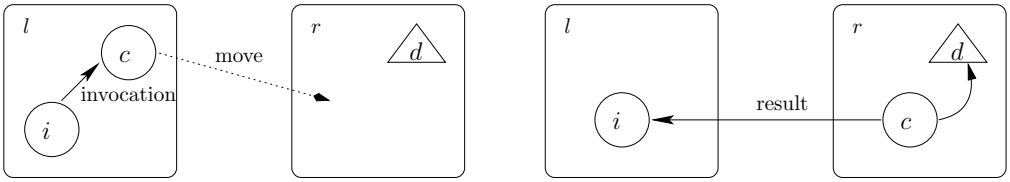
Figure 3.2 depicts all possible single-step moves an invoked component can make relative to its invoker. In Figure 3.2, $h_c \in H$ is component c's host when the call reaches it, and $h'_c \in H$ is the host on which it executes. When $h_c = h'_c$, the move arc becomes a self-loop on h_c, i.e. a null-move, and Figure 3.2 reduces to RPC.

In addition to RPC, three of the possible invocation-triggered code motion patterns implicit to Figure 3.2 are well-known, albeit as "mobile design paradigms," or architectures [16]. In MAGE, we view these patterns as building blocks from which to construct



(a) Before (b) After

Figure 3.3: Code on Demand



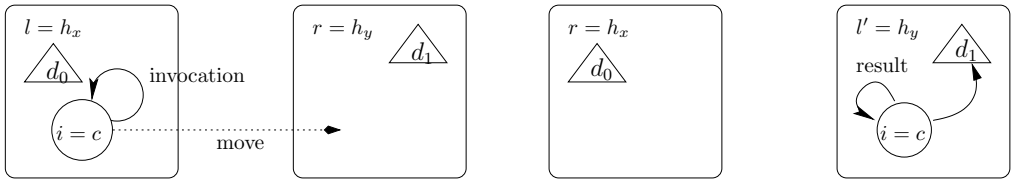
(a) Before (b) After

Figure 3.4: Remote Evaluation

such architectures, not architectures themselves. We describe these three patterns next.

Figure 3.3 depicts *Code on demand* (COD). In COD, a function invocation requires remote code c to execute locally because c requires the resource d. The code moves from the remote site to the caller's site. In the client-server model, COD allows the server to extend the capabilities of the client. Java applets [6], JavaScript [112], and Adobe Flash [110] are popular instances of this mobility pattern. In terms of Figure 3.2, COD occurs when $l \neq h_c \wedge l = h'_c$.

Figure 3.4 depicts *Remote evaluation* (REV) [93]. In REV, the component is initially local to the invoker, then moves to a remote target where it executes and accesses d. In the client-server model, REV allows the client to extend the capabilities of the server. Dispatching an SQL query to a database server is an example of REV. We also used REV in Chapter 2, when we discussed Sun's RMI example and converted it to MAGE. In terms of Figure 3.2, REV occurs when $l = h_c \wedge l \neq h'_c$.



(a) Before (b) After

Figure 3.5: Mobile Agent

Figure 3.5 illustrates *Mobile Agent* [4] (MA) [21, 99]. MA makes most sense when resources are distributed over time or space. Here, c uses both d_0 and d_1. In MA, the invoker calls itself — the invoker is the component. Thus, the component is necessarily active, unlike COD and REV, and l changes as i moves. In terms of Figure 3.2, MA occurs when $l = h_c \wedge l' = h'_c \wedge i = c$.

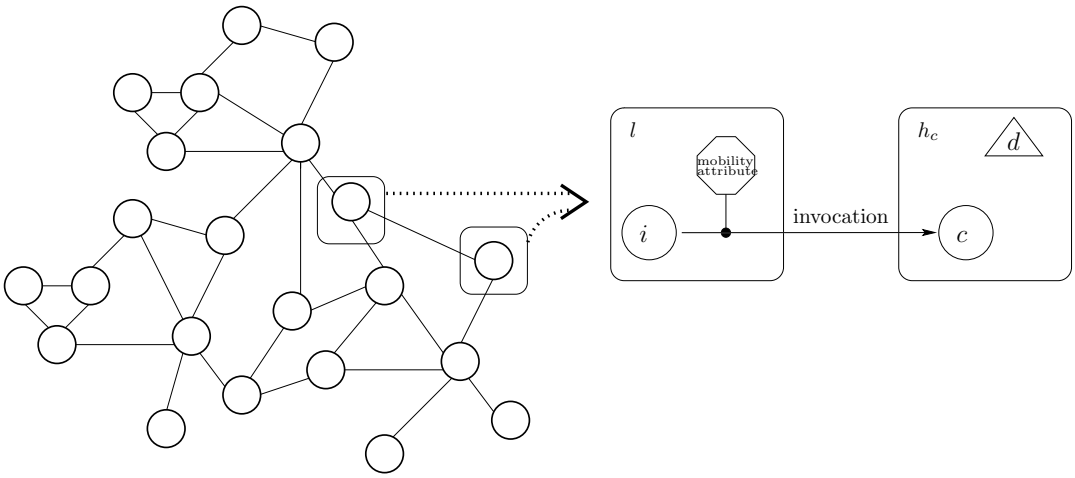
3.3 Primitive Mobility Attributes

At this point, we have exhausted all patterns implicit to Figure 3.2 that have appeared in the literature and been named, not all the patterns it contains. These patterns can be captured as a pair of locations: the location at which a component receives an invocation and the location at which it executes that invocation. These pairs uniquely identify each pattern. In particular, they specify each pattern we have discussed so far. For example, invocation receipt at host r and execution at host l concisely defines COD.

After an invocation, these pairs phenomenologically describe which paradigm a particular invocation used. Specified before an invocation, these pairs can select the paradigm an invocation must use. Distributed systems are complex. Mobile components make them more so. Controlling which paradigm an invocation uses helps the programmer manage that complexity.

Primitive mobility attributes reify these pairs; they specify the host, *start*, at which a

[4] Carzaniga et al. calls this pattern mobile agent. We would prefer to eschew the word agent because of its AI connotations. However, the use of agent in the name of this pattern has gained widespread currency in the literature, so, to avoid confusion, we follow convention. In this thesis, our focus is mobility; whether or not the code is "intelligent" does not concern us.



(a) Program Component Graph (b) Client-side Binding

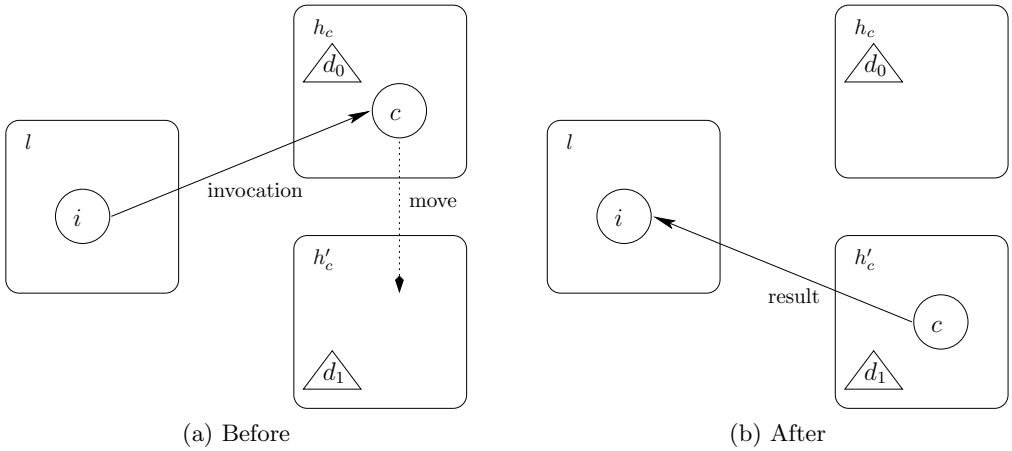
Figure 3.6: Mobility Attribute Binding

component must receive an invocation and the host, *target*, at which it must execute that invocation. Mobility attributes apply before an invocation and control which paradigm an invocation uses. An attribute's start and target pair are constraints that are enforced after an invoker binds a mobility attribute to a component in its namespace. After a mobility attribute has been bound, the component's actual location must match the attribute's starting location or the component does not move, the invocation fails, and the invoker is notified. Upon receipt of an invocation, a component moves to the attribute's target host, when it is not already at that host, before executing.

In Figure 3.6a, nodes are components and edges denote communication. The edges combine the arc from an invoker to a component along which an invocation travels and the corresponding back-arc along which the result returns, as depicted in the pattern figures, such as Figure 3.4 (REV) above. Mobility attributes bind to program components in the context of an invoker and intercept outgoing invocations, as shown in Figure 3.6b.

Definition 3.3.1. *MAGE* is the programming model that incorporates mobility attributes as programmer visible primitives.

Thinking about mobility attributes in terms of before and after pairs helps us see new attributes implicit to Figure 3.2. For $h_x, h_y \in H$ whenever $h_x \neq h_y$ in the pair (h_x, h_y), an



(a) Before (b) After

Figure 3.7: Move Before Execution

invoked component moves. The new attribute *move before execution* (MBE) arises from this observation. MBE generalizes COD, REV, and MA and encompasses all primitive mobility attributes in which a component moves. Figure 3.7 depicts MBE. In terms of Figure 3.2, MBE only requires $h_c \neq h'_c$. From $h_x = h_y$, we form the attribute *execute in place* (EIP) which generalizes LPC and RPC, those attributes in which the component does not move.

Together, EIP and MBE partition the set of all single step component moves in a network relative to an invoker, as depicted in Figure 3.2. If we view the set of mobility attributes as a relation $\mathcal{R}$, EIP is the set of all reflexive pairs in $\mathcal{R}$, i.e. all subsets where if $(x, y) \in \mathcal{R}, x = y$; while MBE is the set of all irreflexive pairs in $\mathcal{R}$, or all subsets where if $(x, y) \in \mathcal{R}, x \neq y$. In MAGE, we assume that EIP is always allowed: that is, the null move (a self-loop) is always allowed.

Attribute	find(id_c) at Invocation	find(id_c) at Execution	Notes
EIP	h	h	
LPC	l	l	$h = l$
RPC	r	r	$h = r$
MBE	h_x	h_y	$h_x \neq h_y$
COD	r	l	$h_x = r \wedge h_y = l$
REV	l	r	$h_x = l \wedge h_y = r$
RMC	r_x	r_y	$r_x, r_y \in R \wedge r_x \neq r_y$

Table 3.1: Primitive Mobility Attributes

Mobility attributes bring into focus and define a taxonomy that encompasses both these new attributes and the mobile code paradigms previously defined in the literature. Table 3.1 depicts this taxonomy.

Table 3.1 does not include MA, because REV and MA are the same with respect to mobility. MA is REV with $i = c$. Thus, MA is restricted to REV on active components, but this hardly seems to merit consideration as a distinct mobility attribute.

At first glance, it might seem strange to see LPC in Table 3.1. It is included because its pair (l, l) is one of the combinations inherent to Figure 3.2. Further, LPC arises naturally in distributed applications. Consider invocations on a component collocated with an invoker via COD. Every invocation other than the first one is LPC; indeed, that is the point of COD. We further motivate LPC below, in Section 3.4.3.

The Notes column in Table 3.1 shows how each attribute specializes either EIP or MBE. The remote mobile code (RMC) attribute is the pair (r_x, r_y), for $r_x, r_y \in R, r_x \neq r_y$. It differs from REV, in that it expects to find its bound component at r_x, not l. In other words, it restricts MBE to motion between two hosts remote to the invoker.

3.3.1 Operational Semantics

The operational semantics for a programming language describes how to interpret a valid program as sequences of computational steps. These sequences then are the meaning of the program. Often these steps are defined in terms of a state transition system that abstractly captures the state of a program as it executes [82, 113]. Operational semantics has two forms. Big-step operational semantics relates programs to a final configuration, ignoring intermediate states. Small-step operational semantics incorporates every state transition in its sequence. Here, we present the big-step operational semantics of primitive mobility attributes.

To model the state of a distributed system, we first project the state of a single machine onto an array. We concatenate all the machine arrays to form an array that represents the state of the entire distributed system. To handle machine arrival and departure, we assume an infinite array and that the block of indices assigned to a machine is never reused:

when a machine returns to operation after going offline, we assign it a fresh name and a fresh range of indices[5]. Since symbolic indices are convenient, let that global state array be associative. When Loc is the set of symbolic indices and Z is the set of cell values, the function $\sigma : Loc \rightarrow Z$ models that associative array. The value of the global state array at x is $\sigma(x) = y$. Any change in the contents of any cell generates a new function σ'. Let Σ be the set of all possible σ functions, or all possible configurations of the global state array, *i.e.* all possible states.

Operational semantics models computation as a sequence of state transitions, or steps. It therefore models time in terms of this sequence. Consider the case of the machine h going offline. Since h is unique, we can use it to store its status — whether it is off- or on-line — in global state. Going off-line then means that at some step h was on-line, then a subsequent step marked it off-line. There is a total ordering of events on a single machine. To order remote events of interest at a particular machine, we use a Lamport clock to define the "happened before" relation [56]. Time is either discrete or continuous. If discrete, we can impose a total ordering on events by ordering the machines, a la Lamport. If continuous, events have a natural total ordering: Let p be the continuous probability mass function of an event occurring and a and b be distinct events on distinct machines. Let $t_a, t_b \in \mathbb{R}$ denote the times at which a and b occurred. Then the probability that t_a equals t_b is 0 since $\int_{t_a}^{t_a} p(t_b)\, dt_b = 0$.

Definition 3.3.2 (Function Overriding)**.**

$$\sigma[x := n](x) = n$$
$$\sigma[x := n](y) = \sigma(y)$$

Changing state involves transitioning from one global configuration to another, from σ to σ', where $\sigma, \sigma' \in \Sigma$. When $\sigma(x) = y$ and a transition changes the value at index x, we employ Definition 3.3.2 and write $\sigma' = \sigma[x := z]$.

[5] Section 6.3.4 describes how the MAGE implementation handles this problem.

Recall that C is the set of all components and that H is the set of hosts[6]. Let $c \in C$ and $h \in H$. Typically, a component is not collocated with an invoker. Thus, an invoker must use a unique identifier to refer to and find a component. Let I be the set of component identifiers. MAGE uses I to query the global state to find a component's current location, so $I \subset Loc$. The function find : $I \rightarrow H$ takes a component identifier and returns that component's current location. The function reads global state, so $I \subset Loc, H \subset Z$ and find $\subset \sigma$. Figure 3.11 defines the semantics of find.

Definition 3.3.3 (Primitive Mobility Attribute)**.** A *primitive mobility attribute* is an element of the set $A = \{(s, t) \in H \times H\}$.

Remark. For $(s, t) \in A$, s is the host at which the bound component must receive the invocation; that is, find$(id_c) = s$ must hold when c is invoked. The host t is the host at which the bound component must execute; that is, find$(id_c) = t$ must hold when c executes.

MAGE is a library that extends a host language. Let $\mathcal{L}$ denote that language. MAGE requires $\mathcal{L}$ to implement some form of RPC; that is, $\mathcal{L}$ must support marshaling, a name service, components either as objects or closures, and define P, a set of RPC proxies for components. Below, MAGE assumes $\mathcal{L}$ supports exceptions, but this is not essential. MAGE does not extend $\mathcal{L}$'s type system or require $\mathcal{L}$ to provide any other features, not already mentioned. We do not formally specify the semantics of a proxy-mediated RPC call, but assume it. Informally, a proxy is generated for a component and has the same interface as that component. When an operation is invoked on a proxy, the proxy marshals the operation's parameters and sends them to its component's host, which in RPC is immutable. If execution generates a return value, the proxy unmarshals it before returning it to the caller.

An RMI proxy contains the host, port pair that identifies its component's invocation server and a component identifier the invocation server uses to route an incoming call to the component. A MAGE proxy additionally contains a mobility attribute binding. That binding may be null, as when no mobility attribute has yet been bound to a MAGE proxy.

[6] As before, C is the vertex set of Figure 1.1a; H is the vertex set of Figure 1.1b.

We denote the null binding as {NULL} and let $A' = A \cup \{\text{NULL}\}$. To add MAGE proxies to the global state, we define $P_m \subset Loc$, $(I \times P \times A') \subset Z$, and bind : $P_m \rightarrow (I \times P \times A')$. Note that bind $\subset \sigma$. Thus, P_m is the set of MAGE proxy variables. An RPC proxy also contains a unique identifier for its component, so the MAGE's proxy's I component does not distinguish the two. The I component in the MAGE proxy is lifted out of its RPC proxy for notational convenience in the rules that follow. The essential difference between a MAGE and an RPC proxy is the inclusion of A', which represents a mobility attribute binding. A MAGE proxy need not always be bound to a mobility attribute. To denote an unbound MAGE proxy, we bind NULL to it. Thus, A' represents all possible bindings of mobility attributes to MAGE proxies, including nonbinding. For the 3-tuple $(I \times P \times A')$, let $\hat{x}_i$ be its i^{th} basis. Initially, no mobility attribute is bound to any MAGE proxy, so $\forall p_m \in P_m,\ \hat{x}_2 \bullet \text{bind}(p_m) = \text{NULL}$[7]. Figure 3.12 contains the rules that define the semantics of binding a mobility attribute to a MAGE proxy.

Definition 3.3.4 (Judgment)**.** The *judgment* $\langle e, \sigma \rangle \Downarrow n$ means that e evaluates to n in state σ. We can interpret $\Downarrow$ as a function with arguments e and σ.

In operational semantics, an evaluation rule is analogous to the Fitch format in logic, which divides an argument's premises from its conclusion using a horizontal line, called the Fitch bar. The premises are written above the line, the conclusion below [29]. In operational semantics, the judgments above the bar are *hypothesis judgments* that must hold before we can infer that the *conclusion judgment* below the bar holds.

When a rule has no judgment above the bar, it is a primitive evaluation rule that always holds.

There are two ways to read an evaluation rule — forward, starting with the hypothesis judgments, and backward, starting with the conclusion judgment. In the forward direction, if we know that the hypothesis judgments hold, then we can infer that the conclusion judgment holds. In the backward direction, they provide a mechanism for evaluating an expression. Consider Figure 3.13a which contains the evaluation rule that defines the semantics of a

[7]Here, we use the i^{th} unit basis vector to extract the i^{th} component. Recall that Kronecker delta $\delta_{ij} = 1$ if $i = j$ and 0 if $i \neq j$. Since $\delta_{ij} = \hat{x}_i \bullet \hat{x}_j$, $\hat{x}_i \bullet \vec{v}$ is the i^{th} component of $\vec{v}$.

$$
\begin{array}{lll}
com ::= & \mathcal{L} \text{ commands} & \\
& \mid \text{bind}(e, e) & \\
& \mid \text{unbind}(e) & \\
& \mid \text{move}(e, e) &
\end{array}
\qquad
\begin{array}{lll}
e ::= & \mathcal{L} \text{ expressions} & \\
& \mid h & \text{for } h \in H \\
& \mid id_c & \text{for } id_c \in I \\
& \mid p_m & \text{for } p_m \in M \\
& \mid (s, t) & \text{for } (s, t) \in A' \\
& \mid \text{get}(e) & \\
& \mid \text{find}(e) & \\
& \mid \text{proxy}(e) &
\end{array}
$$

Figure 3.8: MAGE Grammar Extensions to $\mathcal{L}$

$$\frac{}{\langle h, \sigma\rangle \Downarrow h : H} \qquad \frac{}{\langle p, \sigma\rangle \Downarrow p : P} \qquad \frac{}{\langle (s, t), \sigma\rangle \Downarrow (s, t) : A'} \qquad \frac{}{\langle id_c, \sigma\rangle \Downarrow \sigma(id_c) : H}$$

$$\frac{}{\langle p_m, \sigma\rangle \Downarrow \sigma(p_m) : I \times P \times A'}$$

Figure 3.9: Primitive Evaluation Rules

$$\frac{\langle e, \sigma\rangle \Downarrow id_c}{\langle \text{proxy}(e), \sigma\rangle \Downarrow \sigma(p_m) : I \times P \times A'}$$

Figure 3.10: MAGE Proxy Generation Rule

MAGE proxy-mediated invocation when no attribute is bound to the MAGE proxy p_m. In the forward direction, we see that if no attribute is bound, $\langle p_m, \sigma\rangle \Downarrow (id_c, p, \text{NULL})$, and a standard RPC invocation on p_m's internal RPC proxy p transitions global state from σ to σ', $\langle p.f(E), \sigma\rangle \Downarrow \sigma'$, then we can conclude $\langle p_m.f(E), \sigma\rangle \Downarrow \sigma'$, that a MAGE proxy-mediated invocation will also transition the global state from σ to σ'. In the backward direction, we see that to evaluate any MAGE proxy-mediated invocation we must first lookup up the current values bound to p_m. If no mobility attribute is bound to p_m, the next and final step in evaluating the MAGE proxy-mediated invocation is to evaluate an RPC call.

MAGE extends $\mathcal{L}$ with the commands and expressions in the grammars in Figure 3.8. In operational semantics, commands change state thus causing the transition $\sigma \rightarrow \sigma'$, while expressions read state, denoted $\sigma(x)$ for the variable x.

Figure 3.9 specifies the rules for evaluating the MAGE primitives. In particular, id_c and p_m are variables. Figure 3.10 depicts the evaluation rule associated with the MAGE proxy

$$\frac{\langle e, \sigma\rangle \Downarrow id_c \qquad \langle id_c, \sigma\rangle \Downarrow h}{\langle \text{find}(e), \sigma\rangle \Downarrow h} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow id_c \qquad \langle e_2, \sigma\rangle \Downarrow h}{\langle \text{move}(e_1, e_2), \sigma\rangle \Downarrow \sigma[id_c := h]}$$

Figure 3.11: Component Location Evaluation Rules

$$\frac{\langle e, \sigma\rangle \Downarrow p_m \qquad \langle p_m, \sigma\rangle \Downarrow (id_c, p, (s, t))}{\langle \text{get}(e), \sigma\rangle \Downarrow (s, t)}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow p_m \qquad \langle e_2, \sigma\rangle \Downarrow (s', t') \qquad \langle p_m, \sigma\rangle \Downarrow (id_c, p, (s, t))}{\langle \text{bind}(e_1, e_2), \sigma\rangle \Downarrow \sigma[p_m := (id_c, p, (s', t'))]}$$

$$\frac{\langle e, \sigma\rangle \Downarrow p_m \qquad \langle p_m, \sigma\rangle \Downarrow (id_c, p, (s, t))}{\langle \text{unbind}(e), \sigma\rangle \Downarrow \sigma[p_m := (id_c, p, \text{NULL})]}$$

Figure 3.12: Primitive Mobility Attribute Evaluation Rules

$$\frac{\langle p_m, \sigma\rangle \Downarrow (id_c, p, \text{NULL}) \qquad \langle p.f(E), \sigma\rangle \Downarrow \sigma'}{\langle p_m.f(E), \sigma\rangle \Downarrow \sigma'}$$

(a) No mobility attribute is bound to p_m

$$\frac{\langle p_m, \sigma\rangle \Downarrow (id_c, p, (s, t)) \qquad \langle \text{find}(id_c), \sigma\rangle \Downarrow h \neq s}{\langle p_m.f(E), \sigma\rangle \Downarrow ex}$$

(b) $\text{find}(id_c) \neq s$ at time of invocation

$$\frac{\langle p_m, \sigma\rangle \Downarrow (id_c, p, (s, t)) \qquad \langle \text{find}(id_c), \sigma\rangle \Downarrow s \qquad \langle \text{move}(id_c, t); p.f(E), \sigma\rangle \Downarrow \sigma'}{\langle p_m.f(E), \sigma\rangle \Downarrow \sigma'}$$

(c) $\text{find}(id_c) = s$ at time of invocation

Figure 3.13: Invocation Evaluation Rules

generator function. Given an id_c, the MAGE proxy generator produces a fresh MAGE proxy that contains the passed id_c. A MAGE proxy also contains an RPC proxy. For brevity, we assume $\mathcal{L}$ provides a mechanism for creating a proxy from a component, which the MAGE proxy generator uses implicitly to assign p to the MAGE proxy it creates. Finally, it assigns NULL to the mobility attribute field.

Figure 3.11 presents the semantics of find and move which query and modify a component's location; Figure 3.12 defines how the bind and unbind commands associate mobility attributes to proxies, and how the get expression returns a proxy's current binding.

In Figure 3.13, $\mathcal{L}$ defines the semantics of sequence (;) and the semantics of an RPC call, which we denote $p.f$. Let $E = e_0, e_1, \cdots, e_{n-1}$ denote the, possibly empty, list of expressions from which the actuals of a call are derived. Recall Definition 3.1.2: The component c is a nonempty set of functions and their associated state. By definition, both p and p_m must wrap and export the same set of functions. In Figure 3.13, $p.f$ denotes an RPC proxy-mediated invocation of $f \in c$.

3.4 Set Mobility Attributes

Primitive mobility attributes throw an exception when their starting location constraint is not met. What if a programmer wants a component to receive an invocation at one of two hosts or does not care where the component receives an invocation? What if a programmer wishes to dynamically control, at invocation, a component's execution target, such as choosing from among a system's lightly loaded machines?

Set mobility attributes provide just this flexibility: they generalize primitive mobility attributes by lifting the start and target constraints from single hosts to nonempty sets of hosts. A programmer can define these sets in terms of functions that are evaluated at runtime.

By allowing the programmer to define the sets at runtime using functions, MAGE allows the programmer to use mobility attributes to express dynamic layout, or computation migration, policies. For example, a programmer can define an attribute whose start set

$$\overline{\langle (S,T), \sigma \rangle \Downarrow (S,T)}$$

Figure 3.14: Mobility Attribute Primitive Rule

tracks machines scheduled to be offline for maintenance. Once bound to a component, invocations that raise the start exception allow the programmer to write code that moves components off the afflicted system.

In this, MAGE contrasts with other programming models that provide code mobility. These other models either restrict a program to a small set of static migration policies, or allow the programmer to express arbitrary migration policies, but at cost of writing such policies themselves. For example, Java provides RMI, a form of RPC, and COD, but restricted to inactive components. By default, Java restricts programmers to choosing between static layout or the code migration policy that COD provides — "execute locally."

Using sets for the start and target allows MAGE to introduce a continuum of constraints from all hosts to a single host, i.e. from "don't care" to a specified host. Indeed, set mobility attributes are a strict superset of primitive mobility attributes. A mobility attribute whose start set and target sets have only a single element is a primitive mobility attribute. For this reason, unless otherwise specified, henceforth, we abbreviate set mobility attribute to simply "mobility attribute."

3.4.1 Operational Semantics

Definition 3.4.1 (Mobility Attribute)**.** A mobility attributes is an element of the set $A = \{(S,T) \in 2^H \times 2^H\}$, where $S = \{h \in H \mid f_s(h)\}$ and $T = \{h \in H \mid f_t(h)\}$ for $f_s, f_t \in H \rightarrow \text{Boolean}$.

Remark. For $(S,T) \in A$, S is where the bound component must start; that is, $\text{find}(id_c) \in S$ must hold when c is invoked; $t \in T$ is where the bound component may execute; that is, $\text{find}(id_c) \in T$ must hold when c executes. If $S \cap T = \emptyset$, then c must move to $t \in T$. S is a precondition on an invocation. Generalizing the precondition from the single host precondition of primitive mobility attributes to a set of hosts relaxes the precondition. Under

$$\frac{\langle e, \sigma\rangle \Downarrow p_m \qquad \langle p_m, \sigma\rangle \Downarrow (id_c, p, (S, T))}{\langle \text{get}(e), \sigma\rangle \Downarrow (S, T)}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow p_m \qquad \langle e_2, \sigma\rangle \Downarrow (S', T') \qquad \langle p_m, \sigma\rangle \Downarrow (id_c, p, (S, T))}{\langle \text{bind}(e_1, e_2), \sigma\rangle \Downarrow \sigma[p_m := (id_c, p, (S', T'))]}$$

$$\frac{\langle e, \sigma\rangle \Downarrow p_m \qquad \langle p_m, \sigma\rangle \Downarrow (id_c, p, (S, T))}{\langle \text{unbind}(e), \sigma\rangle \Downarrow \sigma[p_m := (id_c, p, \text{NULL})]}$$

Figure 3.15: Mobility Attribute Evaluation Rules

$$\frac{\langle e_2, \sigma\rangle \Downarrow \emptyset : H}{\langle \text{move}(e_1, e_2), \sigma\rangle \Downarrow \sigma} \qquad \frac{\langle e_1, \sigma\rangle \Downarrow id_c \qquad \langle e_2, \sigma\rangle \Downarrow T \qquad \langle \text{find}(id_c), \sigma\rangle \Downarrow h \in T}{\langle \text{move}(e_1, e_2), \sigma\rangle \Downarrow \sigma}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow id_c \qquad \langle e_2, \sigma\rangle \Downarrow T \qquad \langle \text{find}(id_c), \sigma\rangle \Downarrow h \notin T}{\langle \text{move}(e_1, e_2), \sigma\rangle \Downarrow \sigma[id_c := t \in T]}$$

Figure 3.16: Move Operator with Target Set Evaluation Rules

$$\frac{\langle p_m, \sigma\rangle \Downarrow (id_c, p, \text{NULL}) \qquad \langle p.f(E), \sigma\rangle \Downarrow \sigma'}{\langle p_m.f(E), \sigma\rangle \Downarrow \sigma'}$$

(a) No mobility attribute is bound to p_m

$$\frac{\langle p_m, \sigma\rangle \Downarrow (id_c, p, (S, T)) \qquad \langle \text{find}(id_c), \sigma\rangle \Downarrow h \notin S}{\langle p_m.f(E), \sigma\rangle \Downarrow ex}$$

(b) find$(id_c) \notin S$ when c is invoked

$$\frac{\langle p_m, \sigma\rangle \Downarrow (id_c, p, (S, T)) \qquad \langle \text{find}(id_c), \sigma\rangle \Downarrow h \in S \qquad \langle \text{move}(id_c, T); p.f(E), \sigma\rangle \Downarrow \sigma'}{\langle p_m.f(E), \sigma\rangle \Downarrow \sigma'}$$

(c) find$(id_c) \in S$ when c is invoked

Figure 3.17: Invocation Evaluation Rules

this definition of a mobility attribute, the "unspecified" or "don't care" location is H and R is "don't care" which remote.

The redefinition of A changes every appearance of (s, t) in the grammar extensions and evaluation rules defined in Section 3.3.1 must change to (S, T) to reflect the change in the definition of mobility attributes. In particular, the primitive evaluation rule for mobility attributes in Figure 3.9 and the mobility attribute expressions and commands in Figure 3.12 change, as shown in Figure 3.15

As in Figure 3.13, the host language $\mathcal{L}$ that MAGE extends defines the semantics of sequence (;), and the semantics of an RPC call, which we denote with $p.f.\ E = e_0, e_1, \cdots, e_{n-1}$ denotes the, possibly empty, list of expressions from which the actuals of a call are derived. Thus, $p.f$ denotes a proxy mediated invocation of f in the context of the component identified by id_c.

In Figure 3.17, $\langle \text{find}(id_c), \sigma \rangle \Downarrow h \overset{?}{\in} S$ checks whether h, the current location of the component id_c names, meets the constraint imposed by the mobility attribute (S, T). If $h \notin S$, MAGE throws an exception, as the rule in Figure 3.17b specifies. If the rule in Figure 3.17c applies, the component c moves, only if it is not already at a host in T, as defined in Figure 3.16.

3.4.2 Mobility d-Attributes

When we do not care where a component is, we simply find it. This use case motivates the new attributes listed in Table 3.2. The d suffix refers to the fact that these attributes largely "don't care" where the component bound to them is found when invoked. For example, CODd is that form of COD that does not care where the invoked component is, so long as that component is not initially local ($S = H - \{l\}$) but becomes local and executes locally. We collectively refer to these attributes as d-attributes.

When a programmer simply wishes to invoke c and does not care where c executes, we introduce *current location evaluation* (CLE). Under CLE, a component executes an invocation at whichever host it receives an invocation. Like EIP, which it generalizes,

Attribute	At invocation, find(id_c) $\in S =$	At execution, find(id_c) $\in T =$	Notes
CLE	H	H	EIP with $h = \text{find}(id_c) \in H$
FMC	$H - \{h_y\}$	$\{h_y\}$	MBE with $h_x \in H - \{h_y\}$, $h_y \in \{h_y\}$
CODd	$H - \{l\}$	$\{l\}$	
REVd	$H - \{r\}$	$\{r\}$	
RMCd	$H - \{l, r\}$	$\{r\}$	

Table 3.2: Mobility d-Attributes

CLE does not express mobility, but, at the same time, makes sense only in the context of mobile components, which can move and therefore must be found. Figure 3.18 depicts CLE. Symmetrically, when a programmer does not care where c starts, so long as it moves to and executes on the specified host $h_y \in H$, we introduce *find mobile code* (FMC), which is MBE when the host where the component receives an invocation does not matter, so long as that location is not the target execution environment. CLE is a synonym for EIPd; FMC is a synonym for MBEd.

RMCd differs from REVd in that RMCd requires $l \notin S$ in addition to the requirement, implicit to all forms of MBE, that $r \notin S$, i.e. c must move.

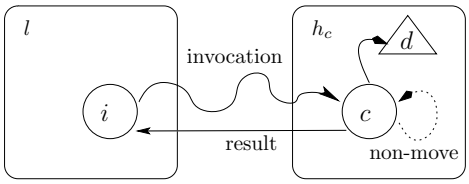


Figure 3.18: Current Location Evaluation

The d-attributes are clearly more flexible than primitive mobility attributes. This fact raise the question: "Why not always use d-attributes?" We address this question next.

3.4.3 Coercion via Mobility a-Attributes

Consider an invocation on the component c, bound to an REV mobility attribute $(\{l\}, \{r\})$ that is already at the specified execution target r[8]. At invocation, the attribute mismatches, since find(id_c) $\notin$ first(REV) $= \{l\}$, but would match at execution, since find(id_c) $\in$

[8]Since set mobility attributes subsume primitive mobility attributes, we incarnate primitive mobility attributes, here REV, in set form.

second(REV) = $\{r\}$. MAGE could either

- invoke that component via RPC; or
- notify the application that REV does not apply.

The former approach emphasizes execution location: since c is already at r, let it execute; who cares how or when c arrived at r? This approach coerces the REV attribute into an RPC attribute. In general, *mobility coercion* handles a starting location mismatch by coercing the mobility attribute into another attribute that has the same execution target and specifies a starting location compatible with a component's actual location if possible, thereby allowing computation to proceed.

The latter approach allows mobility attributes to not only control a program's layout, but to function as *assertions*. Adding mobility (component location) as a primitive to a programming model necessarily makes that model more complex. As assertions, mobility attributes allow a programmer to write code that reacts to a program's layout and enforces layout invariants. For example, when a programmer has an invoker bind LPC to a component, the programmer is asserting that the component should be local to that invoker, and, if not, he wants to be informed. Further, the choice of mobility attribute could reflect lack of trust: a programmer may not want to use COD and localize code from an untrusted server.

We proposed mobility coercion [8] and, in so doing, ruled out using mobility attributes as assertions. This decision also muddied the definition of the application of mobility attributes: when REV coerces to RPC, the actual mobile invocation pattern applied after the computation completes was RPC, not REV. Thus, the attribute that was bound was not applied and vice versa. This thesis makes the opposite choice: we preserve the semantics of mobility attributes as assertions. We do so without losing the convenience of mobility coercion. The discovery of new categories of mobility attributes allows us to capture coercion semantics as attributes!

So the short answer to the question — "why not always use d-attributes?" that we posed above — is "to allow the usage of mobility attributes as assertions." When the programmer

Attribute	At invocation, find(id_c) $\in S =$	At execution, find(id_c) $\in T =$	At invocation, coerces to CLE when
EH	H	$\{h\}$	find(id_c) $= h$
CODa	H	$\{l\}$	find(id_c) $= l$
REVa	H	$\{r\}$	find(id_c) $= r$
RMCa	$H - \{l\}$	$\{r\}$	find(id_c) $= r$

Table 3.3: Mobility a-Attributes

does care about a component's starting location, the programmer can use a primitive mobility attribute or otherwise restrict S to a subset of H; when he does not, he can use a d-attribute. However, the FMC derived d-attributes are all forms of MBE and specify movement; that is, when find(id_c) $= h_x = h_y$, we have CLE, not FMC. Thus, they allow the programmer to specify that a bound componenent *must* move. So, while they are more flexible than primitive attributes, they still have a role as assertions. Rather than coerce d-attributes to CLE generally, and lose their meaning as the assertion that a move must occur, we propose a new class of three attributes each of which coerce to CLE.

Table 3.3 lists these attributes. The category mobility attribute is *execute here* (EH). Each of the listed attributes is the corresponding d-attribute plus explicit coercion to CLE. So, CODa is CODd with explicit coercion; REVa is derived from REVd; RMCa from RMCd; and EH is a synonym for FMCa. By explicitly incorporating coercion, these attributes obviate implicit coercion. We call these a-attributes, from execute at.

To this point, we have focused on set mobility attributes that relax S, the set of valid start locations. CLE is the exception: it relaxes both S and T by setting them both to H. In general, setting $T = Q \subset H$ is useful when one wishes to move the components in the current working set to Q. In the next section, we turn our attention to mobility attributes that dynamically generate S and, especially, T.

3.4.4 Dynamic Mobility Attributes

Static migration policies, while more flexible than static layout, limit a programmer's ability to write code that adapts to its environment, such as to take advantage of new resources or make do with fewer. To demonstrate the semantics of MAGE's support for dynamic

migration policies, we abstractly redefine the `Itinerary` mobility attribute first introduced in Chapter 4.

For any set X, we can define an indicator or characteristic function that indicates whether an element is a member of subset Y of X [111, 86]. Here, we focus on target sets T that change over time. The indicator functions, f_s and f_t, that we introduced in Definition 3.4.1 are a more compact and time-independent way to specify dynamic sets[9].

$$\forall h \in H, f_t(h) = \begin{cases} \text{true} & \text{when } 0 \leq i < n \wedge h = \texttt{itinerary[}i\texttt{++]} \\ \text{false} & \text{otherwise} \end{cases} \tag{3.1}$$

MAGE defines the target set T of its default `Itinerary` in terms of its indicator function $f_t(h)$ as shown in Equation 3.1. An `Itinerary` attribute has an n length array of hosts, `itinerary`. Each invocation of a component through a proxy bound to `Itinerary` advances the index i into `itinerary`. Note that f_t has side-effects, since it depends on the persistence of the index i which counts the invocations of f_t. Figure 4.8, in Chapter 4, depicts the itinerary of c across three invocations, after having been bound to an `Itinerary` mobility attribute whose array is (x, y, z).

In addition to illustrating the utility of dynamically generating T, the `Itinerary` mobility attribute can also naturally capture the expressivity of dynamic S. Consider this use case: the programmer who bound `Itinerary` to c wishes exclusive control over the placement of c and to be notified, via a `StartException`, when c is *not* found at the previous host in the itinerary. Equation 3.2 defines an f_s that accomplishes just this task: at the i^{th} invocation, it restricts c's starting location to the target of the $i-1^{\text{th}}$ invocation, using a trailing index into the target array.

$$\forall h \in H, f_s(h) = \begin{cases} \text{true} & \text{when } 0 = i \wedge h = s \\ \text{true} & \text{when } 0 < i < n \wedge h = \texttt{itinerary[}i-1\texttt{]} \\ \text{false} & \text{otherwise} \end{cases} \tag{3.2}$$

[9] In fact, MAGE's Java implementation defines S and T in terms of their indicator functions f_s and f_t, which simply return S and T when they are static and we suspect that doing so will be convenient in other languages as well.

3.5 Mobility Attribute Operators

As we have seen, mobility attributes can express complex migration policies. To mitigate this complexity, MAGE allows programmers to compose mobility attributes and reuse their migration policies. For example, a programmer can compose the `CPULoadThreshold` attribute with `Evacuate` — a mobility attribute that moves components off systems that are going to be taken down for maintenance — to make a mobility attribute that selects from among those least loaded systems that will remain online.

Since mobility attributes are pairs of sets, we can use set operations to compose them. Given $x = (S_x, T_x), y = (S_y, T_y) \in A$, we can form the mobility attribute $z = (S_x, T_x \cap T_y)$. When T_x specifies systems with light CPU load and T_y contains those systems that have a certain resource, like a DBMS, z is the mobility attribute whose policy is "execute on lightly loaded systems that have a DBMS."

For convenience, MAGE defines operators to directly compose mobility attributes. Set operations are not enough, because a programmer may wish to compose two attributes by relaxing the start sets via union, while taking one target and discarding the other. For this purpose, we define two operators in Equation 3.3: the $\triangleleft$ operator simply returns its left operand, while $\triangleright$ returns its right operand.

$$\begin{aligned} x \triangleleft y &= x \\ x \triangleright y &= y \end{aligned} \tag{3.3}$$

$$\begin{aligned} &\forall (S_a, T_a), (S_b, T_b) \in A, \\ &\quad O \in \{\cup, \cap, \triangleleft, \triangleright\}, \\ &(S_a, T_a)\ O_S O_T\ (S_b, T_b) = (S_a\ O_S\ S_b, T_a\ O_T\ T_b) \end{aligned} \tag{3.4}$$

Equation 3.4 defines the binary operators one can use to compose mobility attributes. For example, $(S_a, T_a) \cup_S \lhd_T (S_b, T_b) = (S_a \cup S_b, T_a \lhd T_b) = (S_a \cup S_b, T_a)$.

To round out its operators, MAGE also lifts set complement to mobility attributes:

$$\begin{aligned} \forall (S_a, T_a), (S_b, T_b) \in A, \\ !_{ST}(S_a, T_a) &= (!S_a, !T_a) \\ !_S(S_a, T_a) &= (!S_a, T_a) \\ !_T(S_a, T_a) &= (S_a, !T_a) \end{aligned} \tag{3.5}$$

Complementing mobility attributes is useful when it is easier to define, especially at runtime, the complement of a set than the set itself. For example, say a particular host h is going off-line. After h goes off-line, H will reflect that fact, but, in the meantime, valid hosts are $!\{h\}$[10]. Also, the addition of complement allows MAGE to define the set difference of two mobility attributes in terms of the operators defined here.

To build intuition about this operators, observe that the $\lhd$ and $\rhd$ operators override one of the operands, $\cup$ acts to relax the constraints to which it is applied, while $\cap$ tightens the constraints on which it operates.

3.5.1 Operational Semantics

MAGE makes these operators available to programs by again extending $\mathcal{L}$'s expression grammar, as Figure 3.19 depicts. Figure 3.20 defines the operational semantics of these operators.

3.6 Component Mobility Attributes

To this point, we have considered only client-side mobility attributes. Here, we apply the client-server paradigm to components with respect to a single invocation: a component

[10] Section 3.3.1 discusses how operational semantics models time.

$$
\begin{array}{rll}
e ::= & \mathcal{L} \text{ expressions} & \\
& \mid e\ o_S o_T\ e & \text{for } o_S, o_T \in \{\cup, \cap, \lhd, \rhd\} \\
& \mid !_{ST}\ e & \\
& \mid !_S\ e & \\
& \mid !_T\ e &
\end{array}
$$

Figure 3.19: Extensions to $\mathcal{L}$'s Expression Grammar for Composition

$$
\frac{\langle e_1, \sigma\rangle \Downarrow (S, T) \qquad \langle e_2, \sigma\rangle \Downarrow (S', T')}{\langle e_1\ o_S o_T\ e_2, \sigma\rangle \Downarrow (S\ o_S\ S', T\ o_T\ T'), \text{for } o_i \in \{\cup, \cap, \lhd, \rhd\}, \text{where } i \in \{S, T\}}
$$

(a) Binary Operators

$$
\frac{\langle e, \sigma\rangle \Downarrow (S, T)}{\langle !_S\ e, \sigma\rangle \Downarrow (!S, T)} \qquad \frac{\langle e, \sigma\rangle \Downarrow (S, T)}{\langle !_T e, \sigma\rangle \Downarrow (S, !T)} \qquad \frac{\langle e, \sigma\rangle \Downarrow (S, T)}{\langle !_{ST} e, \sigma\rangle \Downarrow (!S, !T)}
$$

(c) Unary Operators

Figure 3.20: Mobility Attribute Composition Evaluation Rules

may play both roles — client and server — even simultaneously, over the runtime of an application. Clients, as invokers, define an application's working set and are often active at different times, have different resource needs, and therefore apply different layout policies to a shared server component. Client-side mobility attributes allow different clients to bind different attributes to their view of the same server component, and thereby naturally meet this need.

However, when a development project has multiple teams that produce loosely coupled code or a code base is sufficiently large, the authors of a component may know more about that component's needs when it is invoked as a server than the authors of its clients. To execute correctly, a component, in the server role, may require hosts that have a particular resource, such as a spectrometer. Even when the clients do, or should, know a server component's requirements, server-side mobility attributes are convenient syntactic sugar. When a component's clients can agree on a shared policy, server-side mobility attributes allow programmers to bind a single attribute once in the context of the component, instead of forcing developers to bind clones of an attribute to each client's proxy of that server. Additionally, server-side mobility attributes can specify security policies that restrict where a component moves and executes.

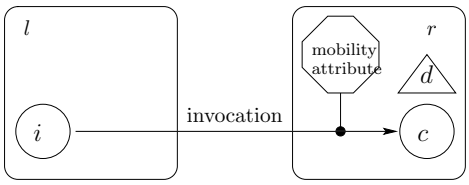


Figure 3.21: Component Mobility Attribute Binding

In this section, we present server-side mobility attributes that bind directly to a component. Figure 3.21 depicts server-side binding of mobility attributes. When a client-side mobility attribute also mediates an invocation, its target set T is part of the invocation message. S does not appear in an invocation message, as its check is enforced at the client and is thus superfluous at the server. MAGE composes T with the component mobility attribute's target set T_c using the operators defined in Section 3.5. Note that find$(id_c) \in S$ holds, or no invocation would have been sent.

Definition 3.6.1 (Component Mobility Attribute)**.** A *component mobility attribute* (CMA) is an element of the set $T_c \in 2^H \cup \{\text{NULL}\}$.

Remark. Unlike client-side mobility attributes, component mobility attributes do not specify a set of valid starting hosts S_c for a bound component. Three observations motivate this difference:

1. Invokers use the S member of a client mobility attribute for assertions; that is, when find$(id_c) \notin S$ or an undesirable component layout configuration holds, MAGE notifies the invoker, which can then react to and correct the problem. The component c can end up on a host that is undesirable from the point of view of an invoker, because that invoker competes with other invokers to place c. In contrast, a component mobility attribute mediates all calls to c, so it can prevent c from ever moving to undesirable hosts. For instance, a programmer can define the CMA $T_c = \{\text{acceptable targets}\}$ and bind it to c using $\cap$ as the mobility attribute operator. Then, no matter what T the client sends, the CMA will cut it down to its acceptable subset $T \cap T_c$.

2. Say we defined CMA to include S_c. Let $find(id) = s$ be the server on which a

$$
\begin{array}{lll}
com ::= & \mathcal{L} \text{ commands} & \\
& \mid \text{bind}(e, e, e) & \\
\\
e ::= & \mathcal{L} \text{ expressions} & \\
& \mid c & \text{for } c \in C \\
& \mid o & \text{for } o \in O
\end{array}
$$

Figure 3.22: Component Mobility Attribute Grammar Extensions to $\mathcal{L}$

component receives an invocation. $s \in S_c$ must hold, or c will be unreachable, since then all invocations will cause c to throw an exception, before ever moving or executing. Thus, when $S_c \neq H$, if c ever moves to $h \in H - S_c$, c becomes unreachable and MAGE provides no way to recover. Therefore, there is a single sensible value for S_c, namely H. Since there is only one sensible value, we implicitly set S_c to H, which obviates the need for a start check and for an explicit S_c parameter.

3. Again assume CMA includes S_c. There is no natural recipient for exceptions thrown when $\text{find}(id_c) \notin S_c$. This is because of the disconnect between the server-side thread that bound a CMA to c and threads that invoke c. The server thread cannot reasonably resolve exceptions raised by the activity of the invoking threads, since it knows nothing about the invoker's context. If, on the other hand, we send the exception to the invoker, the client, who does not know or bind S_c, clearly disagrees since $\text{find}(id_c) \in S$ or the invocation would not have reached c, and, in any case, can do nothing about the problem, since it cannot alter the binding of a CMA to c.

So an S_c starting location check does nothing useful; it only creates new ways to shoot oneself in the foot. Thus, we drop it here.

3.6.1 Operational Semantics

Here, we present the minimal set of rules necessary to define component mobility attributes, as a delta against the rules already presented. The rules below are either new or extend existing rules. Let $O = \{\cup, \cap, \lhd, \rhd, !\} \cup \{\epsilon\}$ be the operators that compose a client's mobility attribute with a component mobility attribute in the server's context. To denote a null binding, $\epsilon \in O$.

$$\frac{}{\langle c, \sigma\rangle \Downarrow \sigma(c) : O \times T_c} \qquad \frac{}{\langle o, \sigma\rangle \Downarrow o : O}$$

Figure 3.23: Primitive Evaluation Rules

$$\frac{\langle e_1, \sigma\rangle \Downarrow c \qquad \langle e_2, \sigma\rangle \Downarrow o \qquad \langle e_3, \sigma\rangle \Downarrow T}{\langle \text{bind}(e_1, e_2, e_3), \sigma\rangle \Downarrow \sigma[c := (o, T)]} \qquad \frac{\langle e, \sigma\rangle \Downarrow c}{\langle \text{unbind}(e), \sigma\rangle \Downarrow \sigma[c := (\epsilon, \text{NULL})]}$$

Figure 3.24: Component Mobility Attribute Evaluation Rules

$$\frac{\langle p_m, \sigma\rangle \Downarrow (id_c, p, \text{NULL}) \qquad \langle c, \sigma\rangle \Downarrow (\epsilon, \text{NULL}) \qquad \langle p.f(E), \sigma\rangle \Downarrow \sigma'}{\langle p_m.f(E), \sigma\rangle \Downarrow \sigma'}$$

(a) No mobility attribute is bound to c

$$\frac{\begin{array}{c}\langle p_m, \sigma\rangle \Downarrow (id_c, p, (S, T)) \\ \langle c, \sigma\rangle \Downarrow (o, T_c) \qquad \langle \text{find}(id_c), \sigma\rangle \Downarrow h \in S \qquad \langle \text{move}(c, T \ o \ T_c); p.f(E), \sigma\rangle \Downarrow \sigma'\end{array}}{\langle p_m.f(E), \sigma\rangle \Downarrow \sigma'}$$

(b) find$(id_c) \in S$ when c is invoked

Figure 3.25: Invocation Evaluation Rules

Figure 3.22 extends the grammar in Figure 3.19 which, in turn, extends the grammar defined in Figure 3.8. It adds operators as terminals and a three parameter bind, where the new parameter is a mobility attribute operator. Figure 3.23 adds these operators as a primitives and associates an operator and component mobility attribute with the component c.

Figure 3.24 makes clear why we need primitive operators: we need to bind them to components along with the component mobility attribute T' to know how to compose T with the incoming T_i that the invoker specified. Here, c is not the proxy p_c, it is the address of a component.

Figure 3.25 specifies how to compose the client mobility attribute's target with the component's attribute. Figure 3.25a handles the case where no CMA is bound to c. Whether or not a component mobility attribute is bound is irrelevant to checking find$(id_c) \in S$, so we

do not repeat that invocation rule here. The move command semantics are unchanged.

Figure 3.25b allows a CMA to override an invoker's T, as when a programmer uses `bind(`c, R, T'`)` to bind and $T \cap T' = \emptyset$. This definition of server-side invocation reduces the client-side mobility attribute's T to a hint that the component is free to ignore. A server component is already free to ignore client invocations and unpublish itself, so giving it control over its mobility does little to change the power relation between it and its clients.

When c executes on $t \in T_c \wedge t \notin T$, the invoker i gets its reply from an unexpected source, and so can discover when its T has been overridden.

3.7 Summary

In this chapter, we have seen how mobility attributes arose from the analytical discovery that existing distributed programming paradigms that integrate invocation and mobility can be abstracted to a pair of hosts, or a primitive mobility attribute. We generalized that pair of hosts to a pair of sets of hosts to form set mobility attributes and showed how these attributes are usefully more powerful than primitive mobility attributes: they obviate coercion, and they facilitate both the composition of attributes and the definition of dynamic, resource-aware attributes, through the definition of indicator functions. We next described how to dynamically define set mobility attributes using their indicator functions, and provided three examples drawn from MAGE's library of mobility attributes to illustrate the point. We introduced operators on mobility attributes that allow one to compose new mobility attributes out of other, simpler attributes. Finally, we presented server-side mobility attributes that allow the authors of components, and not just their users, input into where their components moves and therefore execution.

Chapter 4

The MAGE Programming Model

Man is a tool using animal.

Thomas Carlyle
Sartor Resartus, 1834

In this chapter, we present the MAGE programming model, as realized in Java. We begin with an example.

Email is classically deployed under the client server paradigm. External mail arrives at the server, which delivers to its clients upon request. Unfortunately, a client often does not want much of the mail the server delivers to it. When all of the mail server's clients agree on what constitutes spam, the server can perform the requisite filtering on their behalf. However, the clients may not agree: for instance, I may wish to receive emails from `Orbitz.com`, while you may not. Moreover, my filtering needs may change over time: once I finalize my travel itinerary, I may no longer wish to see emails from `Orbitz.com`. Today's dominant solution to this problem is to use message passing. This wastes network resources: it requires each client to perform their own filtering locally.

Mobile code can optimize and load-balance resource access via collocation. It can also customize behaviour. Given mobility, I can dispatch a personalized filter to the server. In so doing, I trade local work and network utilization for work at the server. Stamos *et al.*

Listing 4.1: Mobility Attribute Usage

```
1 EmailFilterImpl efi = new EmailFilterImpl();
2 EmailFilter ef = (EmailFilter)
3    MageRegistryServer.bind("emailfilter", efi);
4 REV rev = new REV("mailServer");
5 ef.bind(rev);
6 ArrayList mail = ef.getMail();
7 RPC rpc = new RPC("mailServer");
8 ef.rebind(rpc);
```

proposed remote evaluation (REV)[1] to solve such problems [93]. REV occurs when a client sends code to a server for execution.

In MAGE, a programmer uses mobility attributes to control where the components of his application execute in a network. In Listing 4.1, `EmailFilter` is an interface that defines `getMail()`, which returns a filtered list of emails. `EmailFilterImpl` is a mobile class that implements `EmailFilter`. All distributed systems, including MAGE, comprise a registry service that tracks the location of resources, such as mobile objects. Publishing is the act of binding a name to a resource in the registry; it makes the resource available within the system. On lines 2–3, the programmer publishes `efi` by binding it to the name "`emailfilter`" in the MAGE registry. The `MageRegistryServer.bind` call creates and returns `ef`, a proxy to `efi`. On line 4, the programmer creates an `REV` attribute whose execution target is named, imaginatively enough, "`mailServer`," then binds it to `ef` on line 5. The invocation of `getMail()` at line 6 causes `efi` to move to `mailServer` where it executes and returns a filtered set of emails.

Mobility attributes not only change the location of a mobile object, they also make assertions about where that mobile object should be when they are applied. In Listing 4.1, the `REV` attribute asserts that `efi`, to whose proxy the attribute is bound, is initially local. `efi` is no longer local, if the programmer wishes to use `efi` in place at `mailServer`, she must replace `ef`'s binding to an `REV` attribute with a remote procedure call, or `RPC`, attribute, as shown on lines 7–8. If a method were called on `ef` while it was still bound to

[1] REV was first introduced in Section 2.1 and is discussed in detail in Section 3.2.

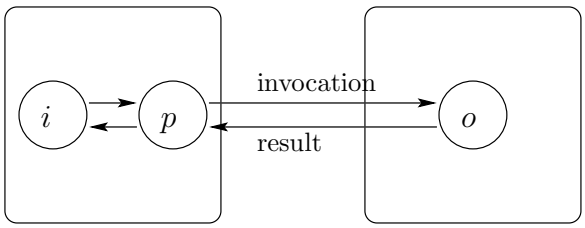


Figure 4.1: RMI Proxy and Remote Object

an `REV` object, but after `efi` had already moved, the call would inform the programmer of the assertion violation by throwing `StartException`. If the programmer simply wanted `efi` to execute at `mailServer` and did not care where `efi` was when it received the call, she could have chosen to bind `REVa`, which expresses exactly this policy: it specifies a target, but imposes no restrictions on mobile object's location when that object receives a call. For more details about `REVa` and related attributes, please refer to Section 3.4.3.

This example introduces three mobility attributes — `RPC`, `REV`, and `REVa`, the mobile class `EmailFilterImpl`, a proxy to an instance of that class, and associates them with bind operators. Below, we explore these primitives and their operators.

4.1 Primitives

To ease deployment, MAGE is implemented as a library. Built on Java, MAGE provides all the usual Java statements and expressions, as well as Java's panoply of types, such as **`int`**, **`char`**, and plain old Java objects (POJOs). To Java's types, MAGE adds mobile objects, proxies to those mobile objects, and mobility attributes.

4.1.1 Mobile Objects

Figure 4.1 describes the relation between a proxy and a remote object. A remote object is an object that can publish itself to a name service and receive remote invocations. An instance of the proxy design pattern [32], a proxy marshals an invocation, including its parameters, and sends it to the remote object.

In RMI, remote objects cannot be marshaled; instead, they are replaced with proxies.

Listing 4.2: `MobilityAttribute`

```
1 public Set<String> starts(Method m) throws StartException
2
3 public Set<String> targets(Method m) throws TargetException
```

POJOs that implement Java's `Serializable` interface can be marshaled, but cannot be invoked remotely. MAGE mobile objects are a modified version of RMI's remote objects that can both receive remote invocations and be marshaled. MAGE moves all the heap objects reachable from the fields of a mobile object. It does not move the stack, registers, or heap reachable from either the stack or registers of a thread executing within that mobile object. Thus, MAGE supports weak mobility [16]. In RMI, a remote class implements Java's `Remote` interface; in MAGE, a mobile class extends `MageMobileObject`.

4.1.2 MAGE Proxy

A MAGE proxy is a Java RMI proxy that supports the binding operations and contains a mobility attribute field.

4.1.3 Mobility Attributes

MAGE is not unique in providing mobile objects; there are many such systems. The mobility attribute is MAGE's unique primitive. As we learned in Chapter 3, a mobility attribute is a pair of sets — `S`, the set of hosts at which a bound object can receive an invocation, and `T`, the target set of hosts at which we want a bound object to execute.

A mobility attribute binds to a mobile object and its proxies. By binding to the proxies of a mobile object as well as directly to the mobile object itself, MAGE allows different invokers to apply different placement policies to their interactions with the mobile object, without incurring the cost of competing over directly binding to the mobile object itself.

We have realized a mobility attribute as an instance of the `MobilityAttribute` base class, shown in Listing 4.2. The `starts` method defines a mobile object's start set `S`; it is an assertion about where a mobile object should be when the mobility attribute is applied. For

Listing 4.3: Method Granular Binding

```
1  @Override
2  public Set<String> targets(Method m) throws TargetException {
3     Set<String>ret = new HashSet<String>();
4     if (method != null && method.equals(m)) {
5        ret = targets();
6     }
7     else
8        ret.add(MageRegistryServer.getLocalHost());
9     return ret;
10 }
11
12 abstract public Set<String> targets() throws TargetException;
```

an application whose objects can move, such assertions are a useful control mechanism. When this assertion is not met, MAGE throws `StartException`. The `targets` determines where the object should move before executing; that is, it defines `T`. Thus, this method is the principal means by which a MAGE application adapts to its environment. As we will see in Section 4.3.3, a mobility attribute can interact with its environment to select a suitable set of execution targets.

Programmers can use the `Method` parameter in both `starts` and `targets` to tie a mobility attribute's behavior to the method invoked, rather than the set of methods defined by an interface. Usually a programmer uses the `Method` parameter to ignore all but a subset of the methods in an interface thus allowing method granular binding of mobility attributes.

Listing 4.3 illustrates how one can use the `Method` parameter of starts and targets to bind to a specific method in an interface. This code is taken from `OneMethodMobAttA`, a class in MAGE's `ml` (for "MAGE library") package which contains all mobility attributes that MAGE defines. Its constructor takes the method instance to which to bind and delegates the actual implementation of the target selection logic to its subclasses via the abstract `targets` method. The if-else forwards the call to `targets` if the call was made on the bound method, otherwise it converts the call into a local procedure call. The if-else can be expanded to select any partition of an interface's methods, not just a single method as shown here.

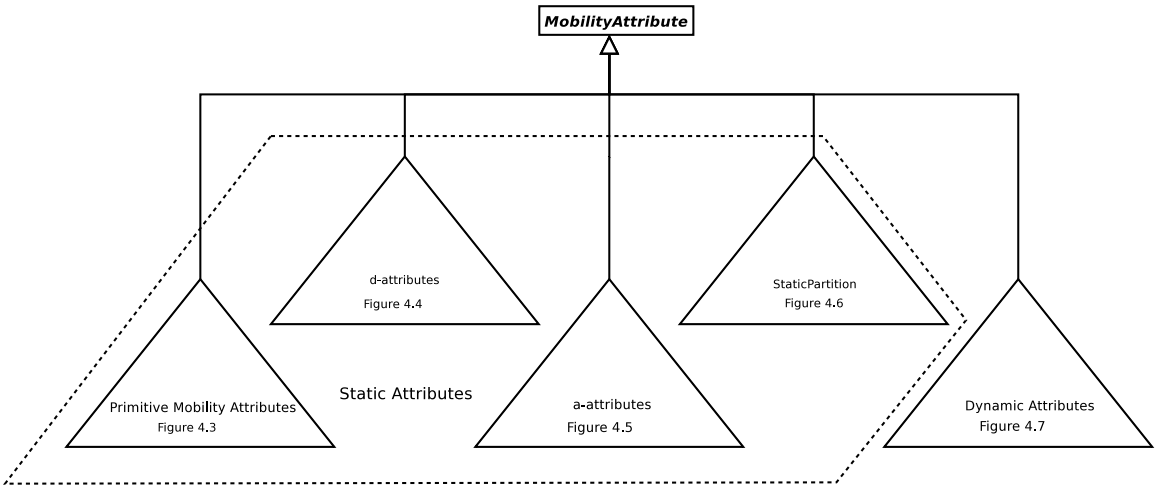


Figure 4.2: Mobility Attribute Class Hierarchy Overview

4.2 Mobility Attribute Class Hierarchy

The MAGE programming model rests upon mobility attributes. In this section, we present the Java class library of predefined mobility attributes MAGE provides. Figure 4.2 presents a high-level overview of the class hierarchy of MAGE's library of mobility attributes. The triangles represent subtrees, which we address in turn with dedicated figures below. Static attributes are those that implement a migration policy that is independent of the runtime environment of the program; while dynamic attributes are those that express migration policies that interact with the environment.

Recall that H is the set of hosts that comprise a MAGE system. For an invoker, $l \in H$ is the invoker's local host and $R = H - \{l\}$ is the set of hosts remote to that invoker. In the figures discussed in the section, $r \in R$ and, in set theoretic terms, `Set<String>` represents the type 2^H.

4.2.1 Primitive Mobility Attributes

Figure 4.3 depicts primitive attributes, described in Section 3.3, implemented as singleton set mobility attributes. The base class `MobilityAttribute` defines the `starts` and `targets` methods. The class `StaticMobilityAttribute` defines an immutable set of valid starting locations `S` and an immutable set of valid targets `T`, then overrides

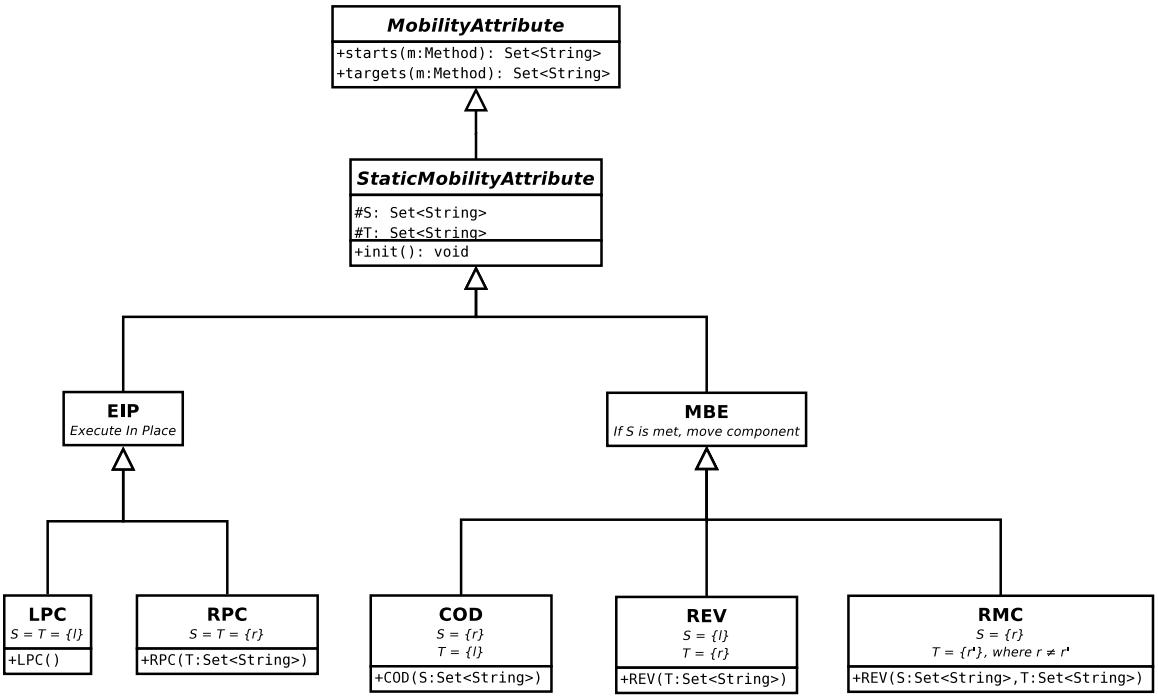


Figure 4.3: Primitive Attributes

`starts(Method m)` to return `S` and `targets(Method m)` to return `T`. The method `init` is abstract in `StaticMobilityAttribute`. `EIP` defines `init` to check that its impose immobility; `MBE`'s `init` similarly checks for mobility.

These attributes do not vary their behavior with resource utilization and layout, but rather unconditionally apply their starting host test and move (or not) their bound object. Although a MAGE application can still use these attributes to dynamically adapt to its environment via judicious binding and rebinding, doing so intersperses logic to manage the application's layout with the application's domain-specific logic. Fortunately, MAGE provides an alternative — dynamic mobility attributes that both isolate layout logic and react to their environment. We describe how to use these attributes in Section 4.3.3.

`EIP` instances are attributes that do not move an object to which they are bound. Here, the two subclasses are assertions. An LPC attribute differs from a local procedure call in that it asserts that a mobile object bound to it is local; an RPC attribute differs from a remote procedure call in that it asserts that a mobile object to it is at the remote host r. If these assertions fail, invocations on the bound object throw `StartException`.

Listing 4.4: The RPC Class

```
1  package ml;
2
3  import java.util.HashSet;
4  import ms.MageServer;
5
6  public class RPC extends EIP {
7     public RPC(String t) throws MobilityAttributeException {
8        String localHost = MageServer.getLocalHost();
9        if (t.equals(localHost))
10          throw new MobilityAttributeException(
11             "RPC: localhost " + localHost + " passed as target"
12          );
13       T = new HashSet<String>();
14       T.add(t);
15       init(T);
16    }
17 }
```

MBE instances, in contrast, are attributes that must move the object to which they are bound, if that object meets their starting host condition. The COD attribute realizes code on demand [16] and requires that a bound object start at r and move to the invoker's host before executing; REV requires that a bound object start at invoker's host and move to the remote target r before executing. The remote move code (RMC) attribute requires the bound object to move between two distinct hosts remote to the invoker. We introduced and discussed the semantics of these attribute in Section 3.2.

A programmer can either instantiate and bind an attribute MAGE provides, such as one in Figure 4.3, or define their own. To do that, the programmer need only extend an attribute class and define the `starts(Method m)` and `targets(Method m)` methods. Their definitions can be quite simple or arbitrarily complicated, as MAGE imposes no restrictions on the bodies of these methods.

Listing 4.4 contains MAGE's definition of the RPC attribute, which we used in Listing 4.1, the example that opens this chapter. RPC's constructor checks that the passed in target location `t` is not local. If it is local, RPC throws `MobilityAttributeException`, the

Listing 4.5: `EIP`'s `starts(Method m)` and `targets(Method m)` Methods

```
1 public Set<String> starts(Method m) {
2    return S;
3 }
4
5 public Set<String> targets(Method m) {
6    return T;
7 }
```

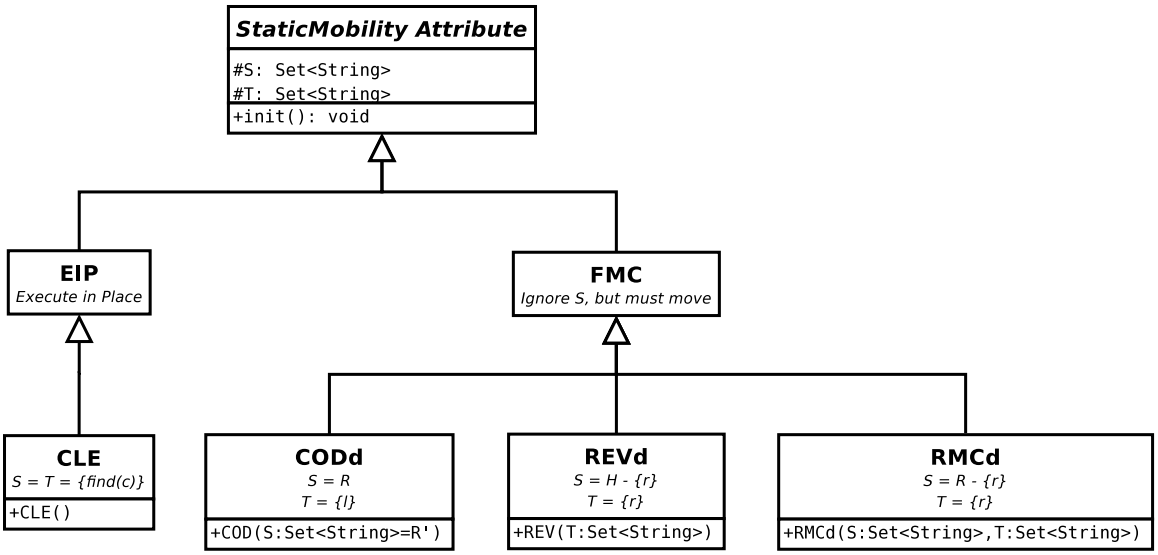


Figure 4.4: d-Attributes

base class of `StartException` and `TargetException`. Then constructor adds `t` to `T`, which `RPC` inherits from `StaticMobilityAttribute`. The `init` method is inherited from `EIP` and sets `S` equal to `T`, to enforce nonmovement.

`RPC` inherits its `starts(Method m)` and `targets(Method m)` methods from `EIP`. Listing 4.5 gives their definitions. `EIP` instances simply return the field, a set of strings, associated with the valid starting locations or desired targets, as appropriate.

4.2.2 The d- and a- Mobility Attributes

Figure 4.4 depicts d-attributes, described in Section 3.4.2. The d-attributes differ from the primitive attributes in that they relax the starting location requirement embodied by `S`. The `FMC` subclasses still require movement to occur. The a-attributes address this issue: a

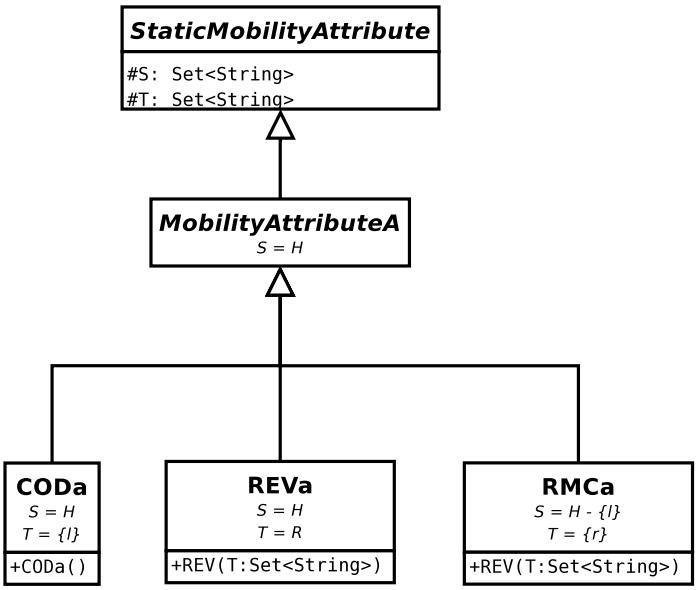


Figure 4.5: a-Attributes

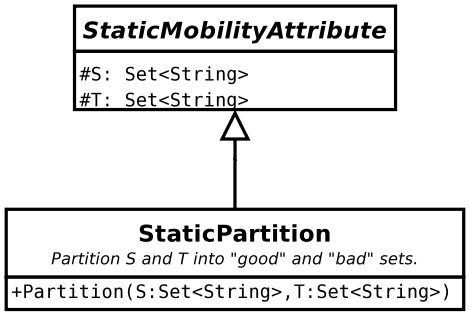


Figure 4.6: `StaticPartition`

programmer uses them when she does not care where a component starts out or whether it moves, just so long as it executes on the target. These attributes effectively ignore `S` by setting it to `H`, and are semantically equivalent to explicitly moving the invoked component to the execution target prior to its execution. Figure 4.5 depicts the a-attributes, which were introduced in Section 3.4.3.

Figure 4.6 depicts the `StaticPartition` attribute. Unlike primitive, d-, and a-attributes, this static attribute is not derived from distributed invocation paradigms. The `StaticPartition` attribute bipartitions H. When a programmer creates an instance of `StaticPartition` in which $S = T$ and binds it to a component, he is asserting that

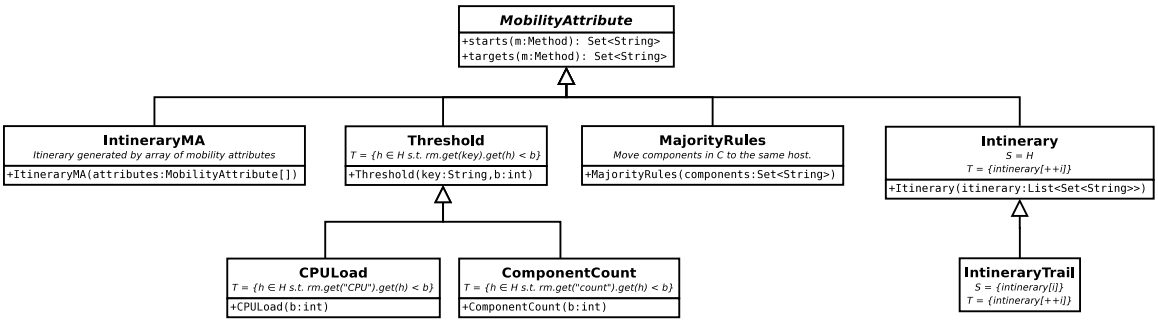


Figure 4.7: Dynamic Mobility Attribute Class Hierarchy

that component *is* in S. When a programmer creates an instance of `StaticPartition` in which $S = H \neq T$, she forces components bound to that instance to move to T. `StaticPartition` is especially useful as a filter used to create a more complicated mobility attribute using mobility attribute operators (Section 4.4.2).

4.3 Dynamic Mobility Attributes

The mobility attributes presented so far immutably define their `S` and `T` sets. In Figure 4.7, we present dynamic attributes, attributes whose S and T sets change.

4.3.1 The `Itinerary` Mobility Attribute

When building a distributed application from mobile objects, programmers often write code to control the route a mobile object takes while moving through the network. So often, in fact, that such code has been identified as the itinerary pattern [101]. MAGE represents the itinerary pattern as an `Itinerary` mobility attribute.

Listing 4.6 contains the definition of MAGE's `Itinerary` attribute. An `Itinerary` attribute has an `ArrayList<Set<String>>` of hosts, `itinerary`. Each invocation of a mobile object through a proxy bound to `Itinerary` advances the index i into `itinerary`. `Itinerary` binds to all methods in a MAGE interface, and thus ignores its `Method` parameter. Figure 4.8 depicts the itinerary of c, starting from s, across three invocations, after having been bound to an `Itinerary` mobility attribute whose list is (x, y, z). If

Listing 4.6: The `Itinerary` Class

```
1  package ml;
2
3  import java.lang.reflect.Method;
4  import java.util.ArrayList;
5  import java.util.Set;
6
7  import ms.MageRegistryServer;
8
9  public class Itinerary extends StaticMobilityAttribute {
10
11    private static final long serialVersionUID = 1L;
12    protected ArrayList<Set<String>> itinerary;
13    protected int index;
14
15    public Itinerary(ArrayList<Set<String>> itinerary)
16       throws MobilityAttributeException
17    {
18       this.itinerary = itinerary;
19       index = 0;
20       // Set S to the set of all known MAGE VMs.
21       S = MageRegistryServer.getVMs();
22       init();
23    }
24
25    protected void init() throws MobilityAttributeException {
26       if (itinerary == null || itinerary.size() < 1)
27          throw new MobilityAttributeException("Empty Itinerary");
28    }
29
30    @Override
31    public Set<String> targets(Method m) throws TargetException {
32       if (index < itinerary.size())
33          return itinerary.get(index++);
34       else
35          throw new TargetException("Itinerary: array exhausted.");
36    }
37 }
```

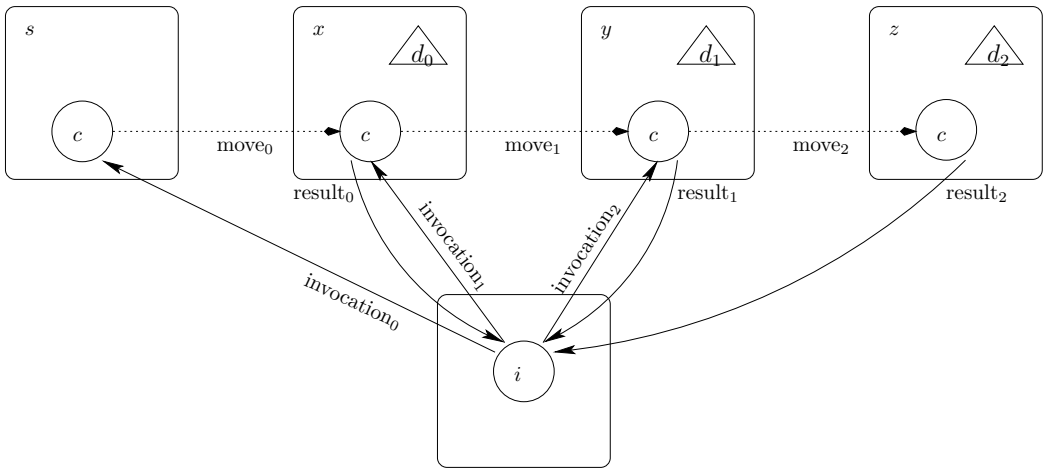


Figure 4.8: Itinerary Mobility Attribute in Action

desired, a programmer can define `Itinerary` instances whose lists contain repetitions, such as (x, x, y, z, z).

`Itinerary` contains a list of sets of hosts. When this is a list of singletons, the attribute realizes a standard itinerary. Since `Itinerary` sets $S = H$, it does not care where its component is before application, and is useful in the context of a component that an invoker shares with other invokers and therefore does not know where the component may be prior to an invocation. When a component is not shared and the programmer wants complete control over the component's itinerary, she can use `FixedItinerary` which defines `starts` to return the last used set in the itinerary array.

`ItineraryMA` takes an array of mobility attributes, through which it steps to form the itinerary. Since any one of its constituent mobility attributes could be dynamic, we have included it here.

4.3.2 The `MajorityRules` Mobility Attribute

A system built from mobile components is susceptible to the problem of moving its components so frequently that the cost of that movement outweighs the benefit of the resulting layout. When this happens, movement impedes useful work and the time-to-completion of services provided by the system suffers. By analogy to page thrashing [23], we call

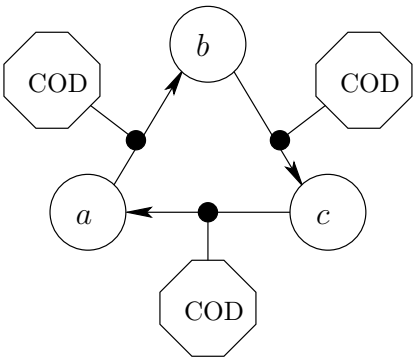
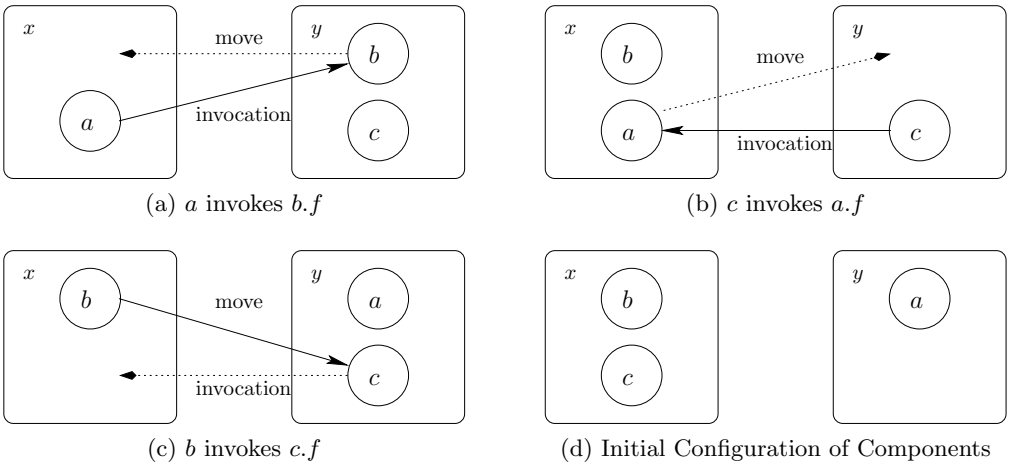


Figure 4.9: Component Control Flow Graph



(a) a invokes $b.f$ (b) c invokes $a.f$

(c) b invokes $c.f$ (d) Initial Configuration of Components

Figure 4.10: Component Migration Thrashing

this problem *migration thrashing*. In the limit, migration thrashing leads to a form of livelock [117] in which the system does nothing but move its components.

Figure 4.9 depicts a scenario that can lead to migration thrashing. The mobile objects $a, b, c \in C$ are interdependent: they represent computations and associated state that should be collocated. Their pairwise dependency manifests itself as a cycle in the call graph. The programmer saw their pairwise interdependence, but not the cycle, and incorrectly bound `COD` to each mobile object's proxy to its neighbor.

Figure 4.10 depicts what happens when `COD` applies and the invocation order $a \rightarrow b.f, c \rightarrow a.f$, and $b \rightarrow c.f$ occurs. In Figure 4.10d, the mobile objects have swapped hosts but are otherwise in the same configuration. As long as these mobile objects are scheduled in this order, migration thrashing will continue.

Listing 4.7: MajorityRules

```
1  package ml;
2
3  import java.lang.reflect.Method;...
4
5  public class MajorityRules extends MobilityAttributeA {
6
7      private static final long serialVersionUID = 1L;
8      protected Set<String> components; // The set of components to collocate.
9      protected Map<String,Integer> countByHost;
10     protected ResourceManagerServer rms;
11
12     public MajorityRules(Set<String> components)
13         throws MobilityAttributeException
14     {
15         if (components == null || components.size() == 0)
16             throw new MobilityAttributeException(
17                 "MajorityRules: invalid component set"
18             );
19         this.components = components;
20         countByHost = new HashMap<String,Integer>();
21         rms = (ResourceManagerServer)
22             MageServer.getComponent("ResourceManager");
23     }
24
25     @Override
26     public Set<String> targets(Method m) throws TargetException {
27         Set<String> ret = new HashSet<String>();
28         Integer value;
29         String h;
30         int max = 1, n;
31
32         countByHost.clear();
33         for (String c : components) {
34             try {
35                 h = MageRegistryServer.find(c);
36             } catch (Exception e) {
37                 throw new TargetException("MajorityRules: find failure", e);
38             }
39             value = countByHost.get(c);
40             if (value == null)
41                 countByHost.put(h, 1);
42             else {
43                 n = value.intValue() + 1;
44                 countByHost.put(h, n);
45                 if (n == max)
46                     ret.add(h);
47                 if (n > max) {
48                     max = n;
49                     ret.clear();
50                     ret.add(h);
51                 }
52             }
53         }
54         return ret;
55     }
56 }
```

MAGE cannot prevent migration trashing in general, but it does allow a programmer to write concise policies that can prevent specific instances of it. For instance, if the programmer had noticed the cycle in the call graph that links a, b and c, he could have have bound the `MajorityRules` instead of `COD` to their proxies to each other.

An instance of `MajorityRules`, Listing 4.7, tracks the location of the set of components passed to its constructor. Its `targets` method finds each component, then returns that subset of hosts with the highest count: thus, its target is a host on which a majority of those components resides. Thus, its application causes a component bound to it to move the host on which that majority resides.

If the programmer had bound `MajorityRules` to the proxies a, b and c use to communicate with each other instead of `COD`, migration trashing would not have occurred. When a invokes $b.f$ in Figure 4.10a, b would remain on host y, since a majority of the set $\{a, b, c\}$ already resides on y. Then c's invocation of $a.f$ would cause a to join b and c on y, where all three mobile objects would remain for the duration of their intercommunication.

An instance of `MajorityRules` constructed with a singleton set realizes an attribute whose policy causes components to which it is bound to collocate with the component in its singleton set. When an instance of `MajorityRules` both tracks and is bound to the same component, its behavior is that of an expensive local procedure call.

4.3.3 The MAGE Resource Manager, or Resource Awareness

MAGE's raison d'être is to allow programmers to dynamically decide where computation in their program should occur and thus how they should combine message passing and code migration to accomplish a task. To this end, MAGE must be resource aware [85] and provide programmers a source of system state that they can query when defining a mobile object migration policy via a mobility attribute.

A great deal of work has gone into load information management systems [71]. MAGE stands on the shoulders of giants and leverages this work: MAGE provides its mobility attributes a generic interface to such systems via its resource manager service. The MAGE resource manager maps strings to arbitrary objects that contain resource data. This service

is globally accessible within the address space of a MAGE host. Since each MAGE host has its own resource manager, the service is inherently decentralized.

Because performance statistics are application-specific and expensive to gather, MAGE leaves the population of the resource manager to the application. To use the MAGE resource manager, a programmer must first populate the service with information about a resource of interest. For example, the programmer could write code that polls Java's JMX API [68] to learn a system's CPU load, then stores it in the local resource manager. MAGE does not synchronize the data stored in resource managers on different hosts. To get information from remote resource managers, the programmer must write code to query the remote resource managers and add the results to the local resource manager. Since it is the programmer's responsibility to populate the MAGE resource manager, the freshness of the data is entirely delegated to the programmer. Section 6.3.7 in Chapter 6 discusses the realization of this service via MAGE's `ResouceManagerServer`.

`Threshold` and its subclasses make use of the MAGE resource manager through their `rm` handle. `Threshold`'s constructor takes a key and a bound, or threshold. `CPULoad` is that subclass of `Threshold` that binds key to "CPU;" while `ComponentCount` binds `Threshold`'s key to "count." Invokers that employ `ComponentCount` cause the target component to be crowd-adverse, to seek hosts with fewer than b other components.

Listing 4.7 in Chapter 4 rather expensively finds its component set upon each invocation. Listing 4.8 depicts `componentCount`, a method an application could use to update the MAGE resource manager with component counts by host. Implicitly, if the map contains no mapping for a host, that host's count is zero. The `name` parameter allows an application to define differents sets of components for use in different instances of `MajorityRules`. An application would need to run `componentCount` periodically for each set of components on each host. The `targets` method of `MajorityRules` would then query the local resource manager, rather then finding the components.

Listing 4.9 defines `CPULoad`, a mobility attribute that relies on the MAGE resource manager. This attribute's `targets(Method m)` accesses the MAGE resource manager through `rms`, a field initialized in its constructor. This attribute defines the policy of

Listing 4.8: Populating the MAGE Resource Manager with Resident Component Counts

```
1  protected void
2  componentCount(String name, Set<String> components) {
3     Map<String,Integer> countByHost =
4        new HashMap<String,Integer>();
5     String h;
6     Integer t;
7     for (String c : components) {
8        h = MageRegistryServer.find(c);
9        t = countByHost.get(h);
10       if (t == null)
11          countByHost.put(h, 1);
12       else
13          countByHost.put(h, t+1);
14    }
15    //overwrite previous binding
16    ResourceManagerServer.put(name, countByHost);
17 }
```

executing on lightly loaded systems. Here, we assume that the application periodically updates the map to which "cpuload" is bound. This map binds host names to the output of the UNIX `uptime` command, which reports the length of the run queue for the past 1, 5, and 15 minutes. In Listing 4.9, the `targets` method queries the MAGE resource service to discover the CPU load of each host. The index field selects which of the three load averages to use in the comparison. `CPULoad` then builds a set that contains those hosts whose selected load average does not exceed the programmer-specified threshold `threshold`, passed into the attribute upon construction.

4.4 Operations

MAGE provides four families of operators over its primitives — find, composition operators, bind, and invocation.

Listing 4.9: `CPULoad`

```
1  package ml;
2
3  import java.lang.reflect.Method;...
4
5  public class CPULoad extends MobilityAttributeA {
6
7      private static final long serialVersionUID = 1L;
8      protected ResourceManagerServer rms;
9      protected int index;
10     protected float threshold;
11
12     public CPULoad(int index, float threshold)
13         throws MobilityAttributeException
14     {
15         assert index < 3;
16         this.index = index;
17         this.threshold = threshold;
18         rms = (ResourceManagerServer)
19             MageServer.getComponent("ResourceManager");
20     }
21
22     @Override
23     public Set<String> targets(Method m) throws TargetException {
24         Set<String> ret = new HashSet<String>();
25         Float[] loads;
26         HashMap<String,Float[]> cpuload =
27             (HashMap<String, Float[]>) rms.get("cpuload");
28         for( String h : cpuload.keySet()) {
29             loads = cpuload.get(h);
30             if ( loads[index] <= threshold ) {
31                 ret.add(h);
32             }
33         }
34         return ret;
35     }
36 }
```

Listing 4.10: The MAGE `find` Operators

```
public static String find(String name);
public static MageMobile lookup(String name);
```

4.4.1 Find Operators

`MageRegistryServer` defines these methods. The name parameter must have the RMI URL format `rmi://[host][:port][/[object]]`, in both find methods in Listing 4.10. In this format, `object` is the mobile object's name and `host:port` is that object's origin

server, the server on which the mobile object was first bound to a registry using the first form of the `bind` method defined in Listing 4.18. Each mobile object's name must be unique throughout a system running MAGE. It is the programmer's responsibility to meet this constraint.

The `find` method returns the current location of the named mobile object as a string in IP-Address:port format. It is primarily used by the MAGE framework itself, during invocation. The `lookup` method returns a proxy to the named mobile object. Each distinct invoker of a mobile object must call this method before it can invoke operations on that mobile object.

A programmer must statically know a mobile object's origin server host, in order to download a proxy from the registry at the well-known port 1099. Embedded in the proxy is the invocation server port of that mobile object's origin server.

4.4.2 Mobility Attribute Operators

Mobility attributes are Java classes, so their methods are Turing-complete. Further, mobility attributes can be composed using the standard Object-Oriented techniques of aggregation and inheritance. These allow us to compose attributes in arbitrary ways, but are usually more powerful than is necessary. We may just want to complement an attribute's target. We may want to select a target from the union or the intersection of the two target sets returned by application of two distinct attributes. Listing 4.11 illustrates the boilerplate the programmer would have to write to intersect the start sets while returning the union of the target sets of two mobility attributes.

Mobility attribute operators obviate such boilerplate. Mobility attributes define sets — `S`, the set of valid locations at which a bound object can receive an invocation and `T`, the target set of hosts at which we want a bound object to execute. Since mobility attributes are essentially sets, we can apply standard set theoretic operators to them. Conceptually, mobility attribute operators leverage this fact to realize a domain-specific language for elegantly combining attributes. Listing 4.12 contains MAGE's `Operator` class which defines the supported operations.

Listing 4.11: Manual Attribute Composition

```
1  package ml;
2
3  import java.lang.reflect.Method;
4  import java.util.Set;
5
6  public class DirectUnion extends MobilityAttributeA {
7
8     private static final long serialVersionUID = 1L;
9     protected MobilityAttribute o1;
10    protected MobilityAttribute o2;
11
12    public DirectUnion(MobilityAttribute o1, MobilityAttribute o2)
13    {
14       assert o1 != null && o2 != null;
15       this.o1 = o1;
16       this.o2 = o2;
17    }
18
19    @Override
20    public Set<String> starts(Method m) throws StartException {
21       Set<String> ret = o1.starts(m);
22       ret.retainAll(o2.starts(m)); // intersection
23       return ret;
24    }
25
26    @Override
27    public Set<String> targets(Method m) throws TargetException {
28       Set<String> ret = o1.targets(m);
29       ret.addAll(o2.targets(m));
30       return ret;
31    }
32
33 }
```

Because a mobility attribute is a pair of sets, we must apply two operators, one for each set, when composing two mobility attributes. The `LEFT` and `RIGHT` operators just return either their left or right operand. These two operators are useful when you want to compose one of `S` or `T`, while ignoring the other. For example, for the two mobility attributes $a = (S, T)$ and $b = (S', T')$, $(a.S$ `LEFT` $b.S', a.T$ `RIGHT` $b.T') = (S, T')$, while $(a.S$ `RIGHT` $b.S', a.T$ `RIGHT` $b.T' = (S', T') = b$. For more detail, refer to Section 3.5 which defines these

Listing 4.12: `Operator` Class

```
package ml;

public enum Operator {
   COMPLEMENT, UNION, INTERSECTION, LEFT, RIGHT;
}
```

operators and their semantics.

Listing 4.13: Mobility Attribute Operations

```
public MobilityAttribute complementS();
public MobilityAttribute complementT();
public MobilityAttribute complementST();
public MobilityAttribute combine(
   MobilityAttribute r, Operator opS, Operator opT
);
```

Listing 4.13 shows the mobility attribute operations that `MobilityAttribute` defines. Rather than create a method for each of the possible combinations of operators applied to each of S and T, MAGE provides the `combine` method, shown in Listing 4.14. The programmer calls this method on the left operand, passes in the right operator, and the appropriate operators in `opS` and `opT`. The `combine` method returns a mobility attribute instance whose `starts` method combines the `starts` methods of its operands using the `opS` and the `targets` methods using `opT`. For the mobility attributes m_0, m_1, the union of their S sets combined with the intersection of their T sets, denoted $m_0 \cup_S \cap_T m_1$, becomes m_0`.combine(`m_1`,Operator.UNION,Operator.INTERSECTION)`. The `complement`* methods are syntactic sugar for calls to `combine`.

Listing 4.14: Generic Operator Application

```
1 public MobilityAttribute combine(MobilityAttribute r,
2       Operator opS, Operator opT)
3 {
4    return new MobilityAttribute(this, r, opS, opT);
5 }
```

Listing 4.15: Manual Example Redux

```
1 MobilityAttribute o1 = new MobilityAttribute();
2 MobilityAttribute o2 = new MobilityAttribute();
3 MobilityAttribute redux =
4    o1.combine(o2, Operator.INTERSECTION, Operator.UNION);
```

Listing 4.16: Intersecting `CPULoad` and `Bandwidth`

```
1  CPULoad ll = new CPULoad(2, 0.5); // See Section 4.3.3.
2  Bandwidth bandwidth = new Bandwith("1MB");
3  MobilityAttribute llBandwidth = ll.combine(
4     bandwidth, Operator.UNION, Operator.INTERSECTION
5  );
6  FarmerImpl fi = new FarmerImpl();
7  Farmer farmer = (Farmer)
8     MageRegistryServer.bind("ebayfarmer", fi);
9  farmer.bind(llBandwidth);
10 try {
11    farmer.ploughEbay();
12 }
13 catch (TargetException e) {
14    // Oops, no host has sufficient CPU and network bandwidth.
15 }
```

Listing 4.15 uses mobility attribute operators to re-implements the composition of attributes defined earlier in Listing 4.11. The `combine` method returns `redux`, a new mobility attribute whose `starts` method returns the intersection of the results of the `starts` methods of `o1` and `o2` and whose `targets` method returns the union of the results of the `targets` methods of `o1` and `o2`.

Listing 4.16 illustrates how a programmer might use mobility attribute operators to combine mobility attributes. `FarmerImpl` is a class that gathers and parses the results of Ebay auctions, before storing them in a database. The `Bandwidth` attribute, instantiated on line 2, is a subclass of `Threshold` specialized to return as targets all systems that are consuming less bandwidth than the threshold passed into its constructor. We defined `CPULoad` in Section 4.3.3 to return systems whose CPU load falls below some user-defined threshold and discussed the implementation of its `targets(Method m)` method in Listing 4.9. The

programmer wants `farmer` to run on lightly loaded systems that also have at least 1MB of available network bandwidth. On lines 3–4, the programmer intersects the target sets of the `CPULoad` and `Bandwidth` attributes to capture this policy in `llBandwidth`, then applies the policy by binding the attribute to `farmer`.

`StaticPartition` becomes interesting in conjunction with mobility attribute operators: Combining an instance of `StaticPartition` with a `CPULoad` instance using the target intersection operator creates an attribute that selects the least loaded host within the specified partition.

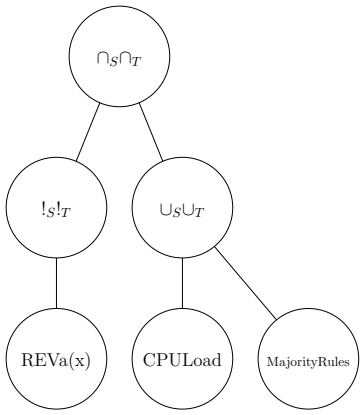


Figure 4.11: Mobility Attribute Operator Tree

Listing 4.17 shows an extended usage example that builds the operator tree shown in Figure 4.11. In the example, we know that a particular system is going down for maintenance. To build a mobility attribute whose target avoids that system, we complement the target of an instance of `REVa`. In general, we want the `c0` component to either run on relatively unloaded systems (1 minute load average less than 2.0) or, to minimize network traffic, on systems where a majority of the components `c0-3` are executing. We realize this policy in the attribute `lowCPUOrMR`. Finally, we intersect `lowCPUOrMR` with `availableHOsts` to ensure that `c0` does *not* run on "hostGoingDown."

4.4.3 Bind Operators

Listing 4.18 contains the three types of binding operators in MAGE.

MAGE borrows the first form of its binding operator from RMI. Defined in the class `MageRegistryServer`, this form binds the mobile object `o` to the identifier `name`, thereby publishing `o` so that it can receive remote invocations. After `o` has bound itself to `name`, clients typically contact the MAGE registry and use `name` to acquire a proxy to `o`. This

Listing 4.17: Operator Tree Example

```
1  REVa adminDown = new REVa("hostGoingDown");
2  CPULoad cpu = new CPULoad("2.0");
3  Set<String> components = new HashSet<String>();
4  components.add("c0"); components.add("c1");
5  components.add("c2"); components.add("c3");
6  MajorityRules mr = new MajorityRules(components);
7  MobilityAttribute availableHosts = adminDown.complementT();
8  MobilityAttribute lowCPUOrMR =
9     components.combine(mr,Operator.UNION,Operator.UNION);
10 MobilityAttribute availLowCPUOrMR = availableHosts.combine(
11       lowCPUOrMR,Operator.INTERSECTION,Operator.INTERSECTION);
12 UIface proxy = MageRegistryServer.lookup("c0");
13 proxy.bind(availLowCPUOrMR);
14 proxy.foo(bar);
```

Listing 4.18: The MAGE Binding Operators

```
public Remote bind(String name, MageMobileObject o);
public void bind(MobilityAttribute ma);
public void bind(MobilityAttribute ma, Operator opT);
```

operator returns a proxy because even in a server context *no* object should have a direct reference to a mobile object, to avoid inadvertently cloning the mobile object and its attendant data coherency problems.

Defined by the `MageMobile` interface, the second form binds mobility attributes to a proxy, so that the attribute can intercept calls mediated by that proxy, as described above in Section 4.1.3.

Defined by `MageMobileObject`, the third form of the bind operator ties a mobility attribute directly to a mobile object, not to a proxy to that mobile object. While a mobility attribute bound to a proxy intercepts a call in the invoker's context, a mobility attribute directly bound to a mobile object intercepts a call in that mobile object's context, at the server on which the mobile object is residing when it receives the call.

In this context, the start set S, and the `starts` method that defines it, make no sense. First, only invocations that meet the invoker's starting location constraint are delivered.

Second, the mobile object's current host must always be in the start set of the attribute directly bound to it, because, if not, every attempt to invoke the mobile object would throw `StartException` and the mobile object would never move or execute. Contrast this with a proxy-mediated invocation, where the invoker can bind a different, more permissive, attribute whose S includes the object's actual location in reaction to receiving a `StartException`. Since a mobile object's current location must always be in the start set of an attribute bound directly to that mobile object, the start set S of directly bound attributes is superfluous and ignored.

The set of targets to which to move the mobile object directly bound to an attribute, is another matter. In MAGE, each time a mobile object is invoked, two targets sets are potentially generated — T_i from the invoker's attribute bound to its proxy and T_d from the attribute directly bound to the invoked mobile object. Rather than simply ignore the invoker's target set T_i, MAGE reuses its attribute composition operators to allow the programmer to decide whether and how to combine T_d and T_i. Allowing the programmer this flexibility is a strict superset of ignoring T_i, as the programmer directly binding an attribute to a mobile object can always select `Operator.LEFT`, thereby causing MAGE to ignore T_i.

With this groundwork laid, we can now unpack and explain this final bind operator's signature. Called on an instance of a mobile object, this bind operator takes a mobility attribute and an mobility attribute composition operator. After the binding, MAGE intercepts invocations on the mobile object in the context of server on which the mobile object is currently residing, generates the target set of the bound mobility attribute T_d, extracts the client's target set T_i from the incoming invocation, then uses the bound operator to combine the two target sets. If the mobile object's current host is in T_d Op T_i, the mobile object executes, otherwise it moves to some $t \in T_d$ Op T_i. The semantics of this operator and component mobility attributes, *i.e.* attributes directly bound to a mobile object, are discussed in more detail in Section 3.6.

Listing 4.19 illustrates the utility of binding mobility attributes directly to mobile objects. `Partition` is an attribute that partitions H, the set of hosts, into "good" and "bad" subsets,

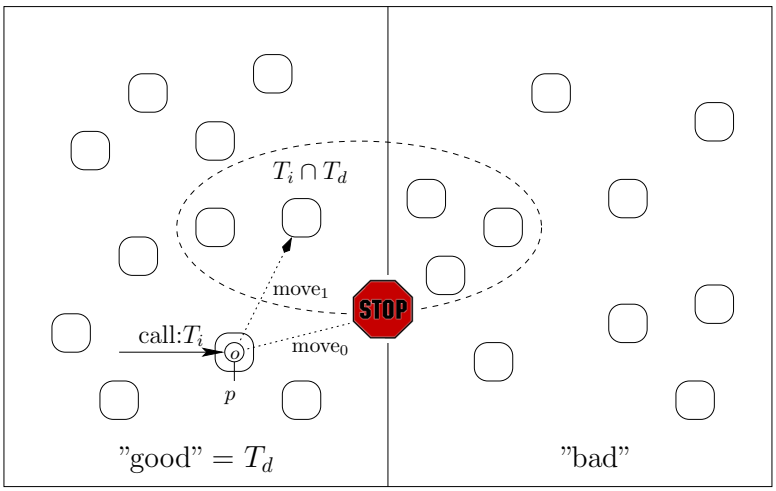


Figure 4.12: Partition Example

Listing 4.19: Binding Component Mobility Attributes

```
1 Partition p = new Partition(goodHosts);
2 MageMobileObject o = new MageMobileObject();
3 o.bind(p, Operator.INTERSECTION);
4 MageRegistryServer.bind("server", o);
```

as shown in Figure 4.12. For instance, a programmer can instantiate a `Partition` attribute that maps hosts that are going down for maintenance to the "bad" set and those that will not to T_d, the "good" set. On line 1, `Partition`'s constructor takes the good subset and instantiates `p`. Let T_i be the target set of the invoker's mobility attribute, embedded in the invocation. On line 3, we bind `p` directly to the mobile object `o` whose execution we wish to restrict to hosts in the "good" partition, using the intersection operation. Subsequently, no matter what T_i is, MAGE restricts `o`'s executions to the intersection of T_d and T_i, that subset of T_i that falls within "good" partition of hosts. In Figure 4.12, move$_0$ is to a host not in the intersection, while move$_1$ is. If the intersection is empty, MAGE informs the invoker with a `TargetException`. Finally, we publish `o` so that it can be invoked on line 4.

4.4.4 Invocation

MAGE proxies are always bound to a mobility attribute. An unbound attribute is implicitly bound to an instance of `CLE`, which instructs MAGE to execute the target mobile object in place, wherever it is. Thus, an invocation on a MAGE proxy first applies the bound attribute to discover its S and T sets. MAGE then finds the component to determine whether its current location is in S. If not, MAGE throws `StartException` to the invoker. Otherwise, MAGE builds and sends the invocation, along with T, to the target component's current location.

The receiving host first checks whether a mobility attribute is directly bound to the target mobile object. If a mobility attribute is directly bound to the target mobile object, the receiving host applies that attribute to produce T', then generates $T_s = T'$ Op T, where T is the client's target set and Op is the operator used in the call to the third form of the bind method above. If no mobility attribute is directly bound to the mobile object, $T_s = T$. If the receiving host is itself in T_s, it simply executes the target component. Otherwise, it adds the target component to the invocation which it forwards to some $t \in T_s$. Conceptually, MAGE invocations implicitly precede invocations with moves.

4.5 Summary

In this chapter, we have explored the MAGE programming model. We have discussed its primitives — mobile objects, proxies, and mobility attributes — and its families of operations over those primitives — find, mobility attribute operators, bind, and invocation.

Chapter 5

Location Services

Attempt the end, and never stand to doubt;
Nothing's so hard but search will find it out.

Robert Herrick, 1591-1674

A distributed system that supports mobile components, such as MAGE, must send messages, in particular invocations, to mobile components as they move. A location service finds resources, such as a mobile component. Dynamic location services include broadcasts, directories, and forwarding pointers. These approaches differ in their lookup and maintenance costs, and their degree of fault tolerance. Forwarding pointers [30] trade lookup for maintenance cost. In this chapter, we show that, for mobile resources, forwarding pointers require fewer messages on average than a centralized directory. Thus, to minimize the number of messages (Section 5.2) and to avoid a bottleneck and a single point of failure, MAGE uses forwarding pointers.

In short, this chapter first juxtaposes forwarding pointers and a centralized directory, then presents and contrasts invocation protocols built using forwarding pointers; it describes the design of MAGE's find operation and its invocation protocol.

5.1 Background

Throughout this chapter, we assume that a network is a fully connected graph.

Definition 5.1.1. A *directory* maps entities, some of which either are or may become remote, to locations.

Remark. An execution environment (host) necessarily knows about local entities. The constraint that some entities must be actually or potentially remote distinguishes a directory from an execution environment.

A directory of mobile components maps the components to hosts. If a network of n nodes uses directories, it can have 1 to n directories. When there are n directories, every system within the network is a directory. A common directory service employs a single, centralized directory (D). Under D, whenever a mobile component moves, it updates its binding at the designated directory. Alternately, we can partition the network or the resources and employ one directory per partition. Each partition, however, reduces to a single, centralized directory in a smaller network. For this reason, we restrict our attention to D.

Under the forwarding pointer location service (FP), every host in the distributed system is a directory, albeit one whose binding for a mobile component may be stale or that may not even have a binding for a given mobile component. Each mobile component starts at some host, which we call its origin. Whenever it moves, it updates its binding in the directory of the host it is leaving to point to the host to which it is moving. This entry is a *forwarding pointer* since it does not necessarily indicate where the mobile component is, but rather that component's last location known to a host.

To statically bootstrap D, the application must statically know the name of the directory. To statically bootstrap FP, the application must statically know the origin of each mobile component, although this burden can be reduced by starting all mobile components at the same host. If there is a single, shared origin, then FP starts out as D. By default, Java uses the former method to statically bootstrap RMI: For each remote object capable of receiving and executing incoming, remote calls, the programmer must statically specify the server on

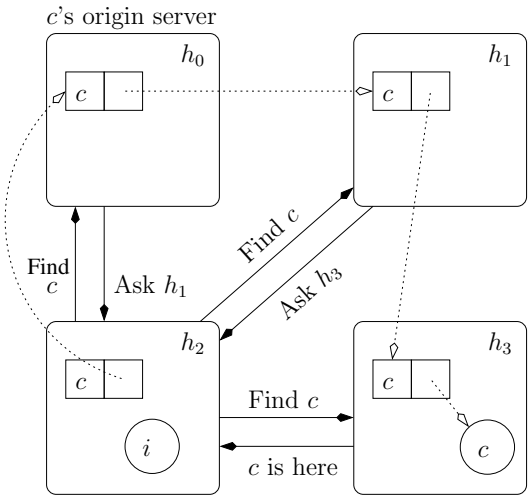


Figure 5.1: Forwarding Pointers

which that remote object resides in order to download a proxy to the remote object. The proxy stores its remote object's server.

To dynamically bootstrap D, one must resort to broadcast — either the directory must broadcast its identity to all hosts or each host must broadcast to discover the directory. FP, in contrast, needs only broadcast until it finds any host that has a forwarding pointer for a particular mobile component.

Figure 5.1 illustrates how a lookup proceeds under FP. Let C be the set of components; let H be the set of hosts and $h_0, h_1, h_2, h_3 \in H$. Assume that c's origin is statically known. For $i, c \in C$, i is searching for c. The invoker i first contacts h_0, c's origin server which refers i to h_1, which was c's destination when c left h_0. Then i contacts h_1 which refers i to h_3 where c resides at the time of the lookup.

D is centralized, while FP is decentralized. Even when FP bootstraps from a single origin server for all mobile components, as components move and lookups occur over time, a system using FP reduces its dependency on that origin server. As a result, D has a single point of failure while, after sufficient movement by mobile objects, FP does not[1]. FP, on the other hand, is more fault-sensitive in that the failure of any host on the path from an

[1]Quantifying "sufficient movement," either by using the Poisson model we describe in Section 5.2 or an application-specific model, is future work.

invoker to an object would prevent that invoker from reaching the object. For clarity, the analysis that follows assumes that machines do not fail.

Of course, we could augment D to use replication or elect a new directory or even fall back to lookup via broadcast. We could use multiple directories, each responsible for a partition of hosts and thereby reduce the number of hosts affected by a directory failure. Replication increases and complicates the cost of updating the information stored in the directory service. Partitioning the network among a set of directories recreates the problem of a single centralized directory within the partition. Thus, D embodies core functionality that these variants either layer upon or reduce to. Therefore, we take D as our point of departure and defer these variants and their analogs for FP to future work.

Initially, D appears superior to FP. For a system with $n = |H|$ hosts, D requires $\Theta(|C|)$ space in the worst case. Under FP, all nodes are directories, so FP requires $O(n|C|)$ space in the worst case. In the worst case, D requires a single lookup message. In FP, the number of lookup messages is the length of a chain of forwarding pointers, or the staleness of the information in the first pointer, and depends on the itinerary of the tracked mobile component. Thus, FP's worst case number of lookup messages is $n - 1$.

Figure 5.2 captures these differences for three location services — broadcast (B), D, and FP. The X-axis is the worst case number of directories in a network. The Y-axis is the worst case number of lookup messages to locate a resource. Again, while that resource is mobile, we assume that it does not move during lookup. We include B as an additional point of reference.

Another way to view the Y-axis is as a measure of the worst case staleness of a particular directory's information. Zero on the Y-axis means no messages were sent to the directory service to find the target resource. This occurs when the searcher and target are collocated or when the searcher's cache is fresh.

B does not require a directory. Thus, B appears on the Y-axis. A broadcast method that sends a single message to each node in a network requires the broadcaster to know every destination. Broadcasting via flooding, in which every node that does not have the queried resource resends the query on every link on which it has not already received a query,

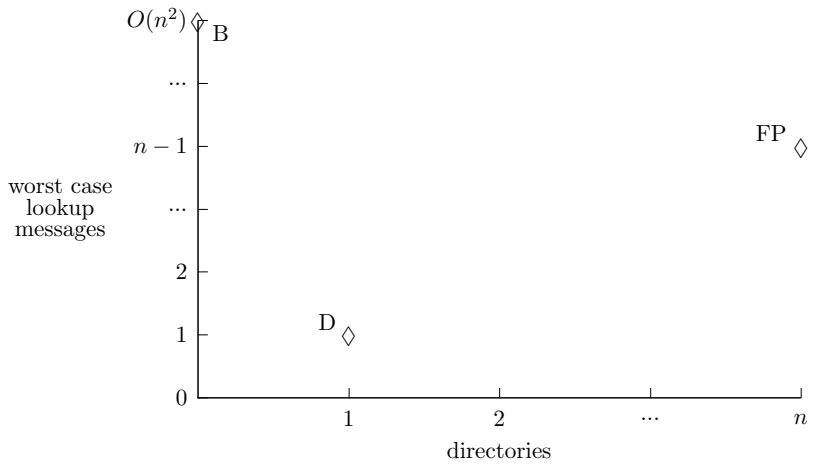


Figure 5.2: Worst Case Location Service Comparison

does not have this requirement [97]. In the worst case, broadcasting via flooding requires messages on the order of the number of edges in the network, as shown.

5.2 Directory vs. Forwarding Pointers: Message Cost

Thus far, it would appear that FP is a non-starter. Consider, however, the message cost of keeping the information in the two location services current. Let m denote the number of moves that the mobile resource made before the lookup under consideration. D requires m update messages, while FP requires 0, because FP builds the cost of update into lookup. Moreover, if the mobile resource returns to a host it already visited, the chain of forwarding pointer is split into two, shorter chains, while the work to update D is unchanged.

To analyze the two location services in terms of their total message costs, we employ the machinery of probability to model the movement of the target resource, so that we can discover the expected length of the FP chain in terms of the number of expected moves of the target resource. In so doing, we dispense with the assumption above that the target resource does not move during lookup.

We compare messages counts because it is independent of timing considerations such as link and host latency. More to the point, we can derive the time cost from messages.

When comparing the message cost of D against that of FP, there are two sources of

randomness — the movement of a target mobile component and the occurrence of searches or invocations, since, in MAGE, searchers are invokers. We use the Poisson distribution in Definition 5.2.1 to model these events [76, p57]. This distribution is continuous and requires that the modeled events be independent. Since $\mathbb{Z} \subset \mathbb{R}$, the use of a continuous distribution is more parsimonious: it does not impose the assumption that sample space be discrete. Our events are the movements of mobile resources and searches for them. A computer program creates, moves, and searches for these mobile objects, so movement and searches are unlikely to be independent in general. In the absence of assuming a particular program and instead thinking in terms of any such program, the assumption of independence for these two variables is, however, a reasonable first approximation of their behavior. Even in the presence of a particular program, this model can serve to measure the degree to which that program's behavior is dependent by comparing the results of this model against the program's observed behavior.

Definition 5.2.1 (Poisson)**.** The probability that k events occur in time t is

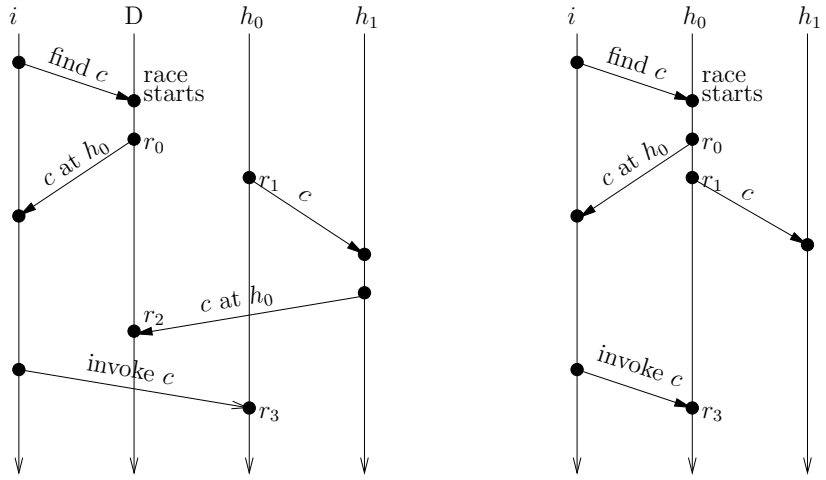
$$Pr(k) = \frac{e^{-\lambda t}(\lambda t)^k}{k!} \tag{5.1}$$

Remark. λ is the expected number of occurrences during t.

When k is the number of moves the mobile component c makes, Equation 5.2 gives the probability that no moves occur in time interval t.

$$Pr(k = 0) = \frac{e^{-\lambda t}(\lambda t)^0}{0!} = e^{-\lambda t} \tag{5.2}$$

Recall that C is the set of components. Without loss of generality, let a searcher be an invoker and $I \subseteq C$ be the set of invokers. Let M be the number of moves the mobile component $c \in C$ makes. Let V be the number of invocations made by $i \in I$. We restrict ourselves to a stream of invocations issued by a single invoker, so our model can incorporate



(a) D Races (b) FP Races

Figure 5.3: Directory Service Races

cache effects. Both M and V are Poisson random variables. Then λ_m is the rate at which a mobile component moves and λ_i is the rate at which an invoker begins its search protocol. We consider M and V over the same period t_e, the period of the experiment.

The mobile component $c \in C$ *moves* when it changes host. This definition does not encompass null moves.

Let r be the rate at which an invoker sends messages and t_r denote a network's round trip time (RTT). The variable t_r is a parameter of the model. By fixing t_r, we ignore variance in RTT, such as congestion or link failure, in order to keep the model simple. If $r > \frac{1}{t_r}$, the invoker issues some messages before it knows the results of previous messages. Further, if $\lambda_m \geq r$, then mobile components are moving at least as fast as invokers are sending messages, and invocations may never complete. Thus, we assume $\lambda_m < r \leq \frac{1}{t_r}$.

Both D and FP are subject to data races, since a mobile component may move at any time. Figure 5.3a contains two races, one between r_0 and r_2 and the other between r_1 and r_3. Both races start when D receives "find c." If r_0 occurs before r_2, then i receives incorrect information. Since r_2 depends on r_1, it depends on the sending and receiving of two messages — c itself from h_0 to h_1 and then the update from h_1 to D. Together, these messages take time greater than or equal to t_r to propagate. If r_3 loses its race with r_1, i's invocation fails.

Event r_3 depends on r_0, so its race too occurs in a period of time of duration at least t_r.

In FP, there is no directory to update, so the two races in Figure 5.3b both involve r_1 directly. If r_1 occurs before "find c" arrives at h_0, then c moves before the invoker is told that c is on h_0, and i just follows another link in the chain of forwarding pointers. To i, this case is indistinguishable from a longer chain. If r_1 occurs after r_0 but before r_3, then c has the time until the find reply reaches and i and i's invocation arrives at r_3 to move, or RTT.

Therefore, the period of time during which these races can occur has duration greater than t_r. From Equation 5.2, the probability that a message wins the race with a mobile component is $e^{-\lambda_m t_r}$, for t_r = RTT, because this is the probability of 0 moves after a successful find reply is sent and the message arrives. For simplicity and because our concern is to compare D and FP, we do not model the time between events occurring on a single system and assume that it is simply zero.

The analysis that follows does not include either the cost of movement, nor the cost of sending an invocation. For concision, we ignore replies to lookup messages, and assume lossless communication.

5.2.1 D's Message Cost

Whenever a mobile component moves, it must update the directory. Thus, D's message cost per move is 1. A lookup may fail due to racing with a move. Equation 5.3 calculates the expected number of find messages, $\langle D \rangle$, that must be issued given that races are possible[2]. Its first line captures the expected cost of a mobile component moving in each race window infinitely often as an infinite series. At the start of each race window, we send a message, the 1 in both terms. If the mobile component does not move during the race window, we have found the mobile object. If it does move, we begin a new race window.

In Equation 5.3, a scripted uppercase letter followed by a colon, such as $\mathcal{A}$:, is a mechanism for implicitly introducing a variable into an equation. Here, we use $\mathcal{A}$ to make algebraically manipulating an infinite series more tractable.

[2] Angle brackets denote the expected value of a random variable.

$$
\begin{aligned}
\langle D_r \rangle &= \mathcal{A} : P(0 \text{ moves in race window})1 + P(j > 0 \text{ moves in race window})1[\\
&\quad P(0 \text{ moves in race window})1 + P(j > 0 \text{ moves in race window})1[\cdots]] \\
\mathcal{A} &= e^{-\lambda_m t_r} + (1 - e^{-\lambda_m t_r})\mathcal{A} \\
\mathcal{A} - (1 - e^{-\lambda_m t_r})\mathcal{A} &= e^{-\lambda_m t_r} \\
\mathcal{A}(1 - (1 - e^{-\lambda_m t_r})) &= e^{-\lambda_m t_r} \\
\mathcal{A}e^{-\lambda_m t_r} &= e^{-\lambda_m t_r} \\
\mathcal{A} &= 1 \\
\langle D_r \rangle &= 1
\end{aligned}
\tag{5.3}
$$

Assuming that c's location is uniformly distributed across H, $\frac{1}{n}$ is the chance that i and c are collocated and $\frac{n-1}{n}$ is the chance that c is remote to i. D's lookup cost is zero when i and c are collocated. This is the $\frac{1}{n}0$ term in Equation 5.4.

$$
\langle D_l \rangle = \langle D_r \rangle (\frac{1}{n}0 + \frac{n-1}{n}) = \frac{n-1}{n} \tag{5.4}
$$

Caching a resource's last known location is a simple and common optimization. Equation 5.5 defines the expected lookup cost under D in the presence of caching. If the cache is up-to-date, no find message is sent, but, if not, the invocation that was sent becomes a find, which costs 1 message, and then a normal lookup occurs, whose cost is given by Equation 5.4. Together, these terms compose the factor $1 + \frac{n-1}{n}$. Notice that t_i, the time interval parameter, is not t_r; it is the interval between invocations originating at a particular host.

$$\begin{aligned}\langle D_c \rangle &= P(\text{0 moves since last invocation})0 \\ &\quad + P(\text{at least one move since last invocation})(1 + \frac{n-1}{n}) \\ &= e^{-\lambda_m t_i}0 + (1 - e^{-\lambda_m t_i})(1 + \frac{n-1}{n}) \\ &= (1 - e^{-\lambda_m t_i})(1 + \frac{n-1}{n}) \end{aligned} \tag{5.5}$$

Thus, D's expected total message cost for a single invoker is

$$\langle M \rangle + \langle V \rangle (1 - e^{-\lambda_m t_i})(1 + \frac{n-1}{n}) \tag{5.6}$$

5.2.2 FP's Message Cost

When a mobile component moves under FP, it leaves behind a forwarding pointer, which costs no message. However, when an invoker seeks to invoke an operation on that mobile component, it must traverse the chain of forwarding pointers built by that mobile component's itinerary. At the time of the j^{th} invocation, let f_j be such a chain of forwarding pointers from an invoker i to a component c. FP directly handles collocation because $|f_j|$ is free to be zero.

FP updates the invoker's forwarding pointer to point to where a mobile component was found after each invocation, as shown in Figure 5.4. Thus, an FP invocation does not start from the target mobile component's origin on each invocation. This point is crucial to understanding the upper bound: each invocation traverses part of the path formed by c's $\langle M \rangle$ moves. This is because the path starts from the invoker i, not a directory.

$$\frac{\langle M \rangle}{2} \leq \sum_{j=0}^{\langle V \rangle} |f_j| \leq \langle M \rangle \tag{5.7}$$

Equation 5.7 bounds FP. The lower bound occurs when c collocates with i every other move. Note that the lower bound cannot be zero, given our definition of a move as a change of host. The component c can revisit a host during its expected budget of $\langle M \rangle$ moves, thus

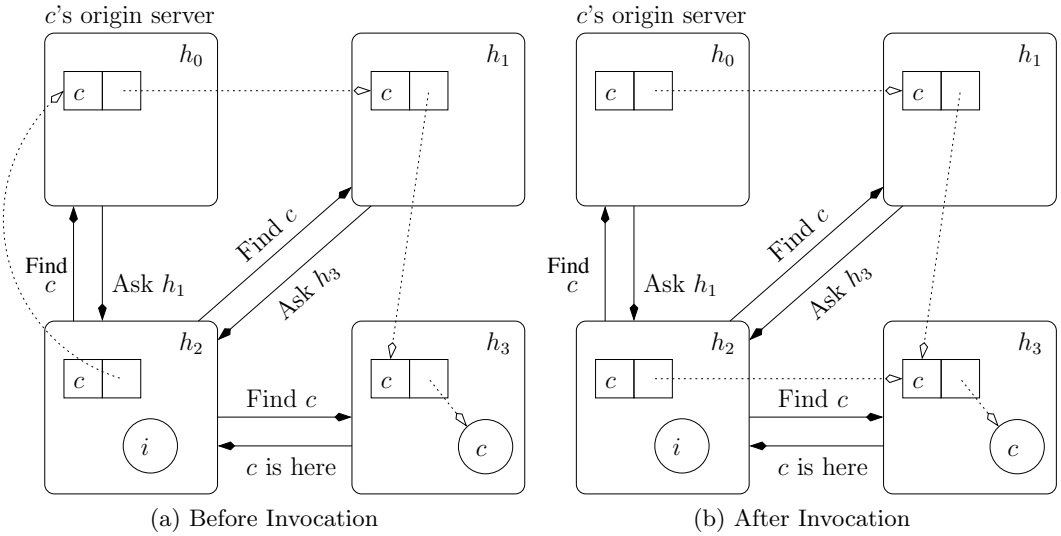
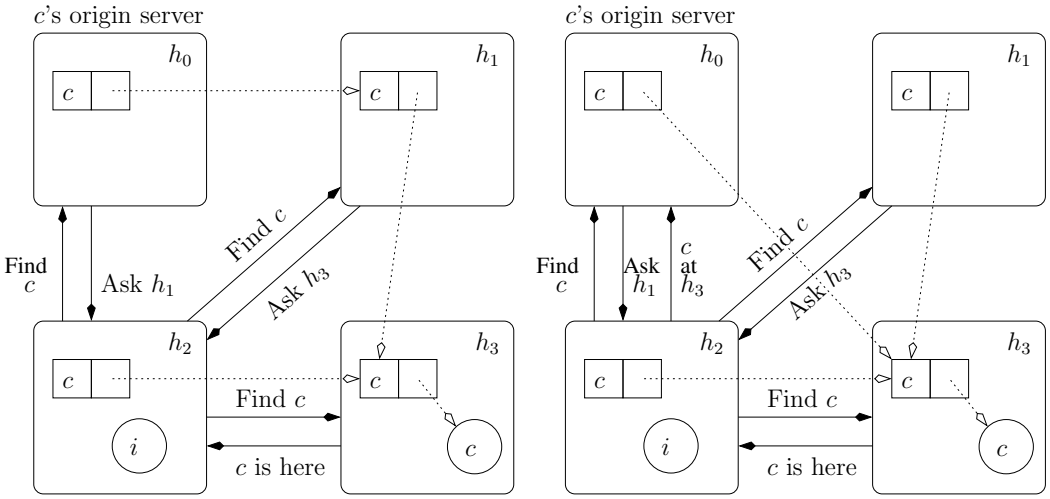


Figure 5.4: Invoker Update Under Forwarding Pointers

splitting its chain of forwarding pointers. When it does so, it shortens the chain for invokers both ahead and behind the revisited node. The longest path c can form is $n-1$. The component c can repeatedly form a longest path by moving $n-1$ times between invocations. When $n < \langle M \rangle$, c necessarily revisits a host.

5.2.3 Collapsing FP Chains

Recall that I is the set of invokers. When $|I| > 1$, D's expected total message cost becomes $\langle M \rangle + |I|\langle V \rangle$, while FP's total becomes $|I| \sum_{j=0}^{\langle v \rangle} |f_j|$. Collapsing FP chains is an optimization that improves the performance of FP when there are many invokers. Because it reduces the expected length of a chain of forwarding pointers, it also reduces the fault sensitivity of FP. FP_c is forwarding pointers with path compression. Under FP_c, an invoker that traverses a chain of forwarding pointers, revisits the hosts along the chain to update their pointers to point to the target mobile component's current host, after it has found the target mobile component. It does not revisit the penultimate hop in the chain or the target component's current host as these hosts have current information. By collapsing the path, an invoker reduces the traversal and update work of subsequent invokers.



(a) Invocation Update (b) Collapse Path

Figure 5.5: Collapse Forwarding Pointer Chain

D	$\langle M \rangle + \langle V \rangle (1 - e^{-\lambda_m t})(1 + \frac{n-1}{n})$
FP	$\sum_{j=0}^{\langle V \rangle} \lvert f_j \rvert$
FP_c	$\sum_{j=0}^{\langle V \rangle} u(j)$

Table 5.1: Directory Service Total Messages

In Figure 5.5, i not only updates its local pointer for c on h_2 to point to h_3 after it has found c, but asynchronously collapses the chain of forwarding pointers to c by updating c's entry on h_0 to point to h_3.

The function $u : \mathbb{N} \to \mathbb{N}$ defined in Equation 5.8 represents the cost of both traversing and collapsing the chain of forwarding pointers $|f_j|$ at the time of invocation j.

$$u(j) = \begin{cases} |f_j| & \text{if } |f_j| \leq 2 \\ 2|f_j| - 2 & \text{otherwise} \end{cases} \tag{5.8}$$

5.2.4 Single Invoker Cost Comparison

Table 5.1 summarizes the expected messages required to both find a component and update the directory service.

$$\langle M\rangle + \langle V\rangle(1 - e^{-\lambda_m t_i})(1 + \frac{n-1}{n}) = \sum_{j=0}^{\langle V\rangle} |f_j| \leq \langle M\rangle$$
$$\langle V\rangle(1 - e^{-\lambda_m t_i})(1 + \frac{n-1}{n}) \leq \langle M\rangle - \langle M\rangle$$
$$\langle V\rangle(1 - e^{-\lambda_m t_i})(1 + \frac{n-1}{n}) \leq 0 \tag{5.9}$$

To compare D and FP for a single invoker, we set their total costs equal and solve. Equation 5.9 implies FP sends fewer messages than D on average, since FP's cost is independent of $\langle V\rangle$, the expected number of invocations, even assuming the worst case movement of the target mobile object, where the adversary makes the invoker perform a traversal for each move. Intuitively, D does a lot of needless work to keep its directory up-to-date: When a mobile object moves three times between invocations, only the last directory update is needed.

Multiple invokers will only strengthen this result for FP, as an arbitrary invoker is likely to benefit from the shortcut performed by another invoker along its path to the object. FP_c only makes sense in the context of multiple invokers. Future work will extend this analysis to multiple invokers.

5.3 Correctness of FP

Here, we assume that moves are instantaneous. In practice, we approximate this assumption by tolerating inconsistency in the tree forwarding pointers for a bounded period of time, see Section 6.5.4. Our proofs use induction on states, so there can be no simultaneous transitions. In particular, we assume a total ordering over message arrival within the network. We also assume that r is not moving faster than the finds that are chasing it. In practice, we enforce this assumption with a bound on the length of the path to a resource that a finder follows before it abandons the search.

Let R denote a set of mobile resources. The *origin* of $r \in R$ is the host at which r enters the network. Let F be the set of finders. $F \neq \emptyset$. Let $H_F \subseteq H$ be the set of hosts on which

finders reside. Let $fp : H \to H$ return a host's forwarding pointer to the resource r.

Initially, Equations 5.11 and 5.12 hold. The nil in Equation 5.12 means that the host does not know how to find r.

$$fp(h) = \text{origin} \tag{5.10}$$

$$\forall h \in H_F, fp(h) = \text{origin} \tag{5.11}$$

$$\forall h \in H - H_F, fp(h) = \text{nil} \tag{5.12}$$

Recall Function Overriding, Definition 3.3.2:

$$\sigma[x := n](x) = n$$

$$\sigma[x := n](y) = \sigma(y)$$

Below we use the prime operator from Lamport's Temporal Logic of Actions [57].

FP Rules

1. **Success**: A search ends when it reaches h_r, r's current host.
2. **Move**: For $h_s, h_d \in H$, when r moves from h_s to h_d, $fp' = fp(h_s := h_d, h_d := h_d)$.
3. **Local Update**: When a searcher on $l \in H$ finds r at $h \in H$, $fp' = fp(l := h)$, *i.e.* the searcher updates its local forwarding pointer to h_r, the host at which it found r.

Let $H_k = \{h \in H \mid fp(h) \neq \text{nil}\}$ and $E = \{(x, y) \in H \times H \mid (fp(x) = y) \wedge (x \neq y) \wedge (fp(x) \neq \text{nil})\}$.

Definition 5.3.1. A *directed tree* is a digraph whose underlying graph is a tree. A *rooted tree* is directed tree with a root and a natural edge orientation either toward or away from the root [116].

Below, we consider only rooted trees whose arcs are oriented toward their root; our root is the vertex reachable from all other vertices. Thus, the root is Rome and "Tutte le strade conducono a Roma" (All roads lead to Rome).

Theorem 5.3.1. *After $n \geq 0$ mutations, $\forall r \in R, G = (H_k, E)$ is a directed tree rooted at h_r, r's current host.*

Proof by Induction.

Base case, $n = 0$: G has $|H_k|$ nodes. r is at origin, so there is no forwarding pointer at origin, since $(\text{origin}, \text{origin}) \notin E$. $\forall h \in H_k - \{\text{origin}\}, fp(h) = \text{origin}$. Thus, G has $|H_k| - 1$ edges and is connected. Therefore, it is a tree. Each edge is an arc whose terminus is origin, so it is a directed tree rooted at h_r.

Inductive step: Assume after n tree mutations, G is a directed tree rooted at h_r.

Two FP rules mutate the tree of forwarding pointers — Rule 2, a resource move, and Rule 3, the local update upon completion of a search.

Case Local Update: Let $l \in H$ be the host of the finder $f \in F$. By the inductive hypothesis, G is a directed tree rooted at h_r. Thus the subtree rooted at l is a subtree, as is the subtree rooted at h_r after the l subtree has been removed. Rule 3 forms G' by changing l's parent pointer to h_r, thus directly hanging l subtree from h_r. This change preserves connectivity, the number of nodes, and edges. Thus, G' is a directed tree.

Case Move: When the $n + 1$ tree mutation is a move, let r moves from h_s to h_d, for $h_s, h_d \in H$.

Case $fp(h_d) = \textbf{nil}$: In G, h_s is the root of a directed tree. $G' = (H'_k, E')$ is formed from $fp' = fp(h_s := h_d, h_d := h_d)$, so G' adds a single node h_d which is connected by a single arc, $h_s \rightarrow h_d$ to G. In other words, $|E'| = |E| + 1$ and $|H'_k| = |H_k| + 1|$. Since $|E| = |H_k| - 1$, $|E'| = |H'_k| - 1$. $\forall h \in H_k$, there is a path from h to h_s, by the

inductive assumption. $(h_s, h_d) \in fp'$, so G' is connected. Thus, G' is a directed tree rooted at r.

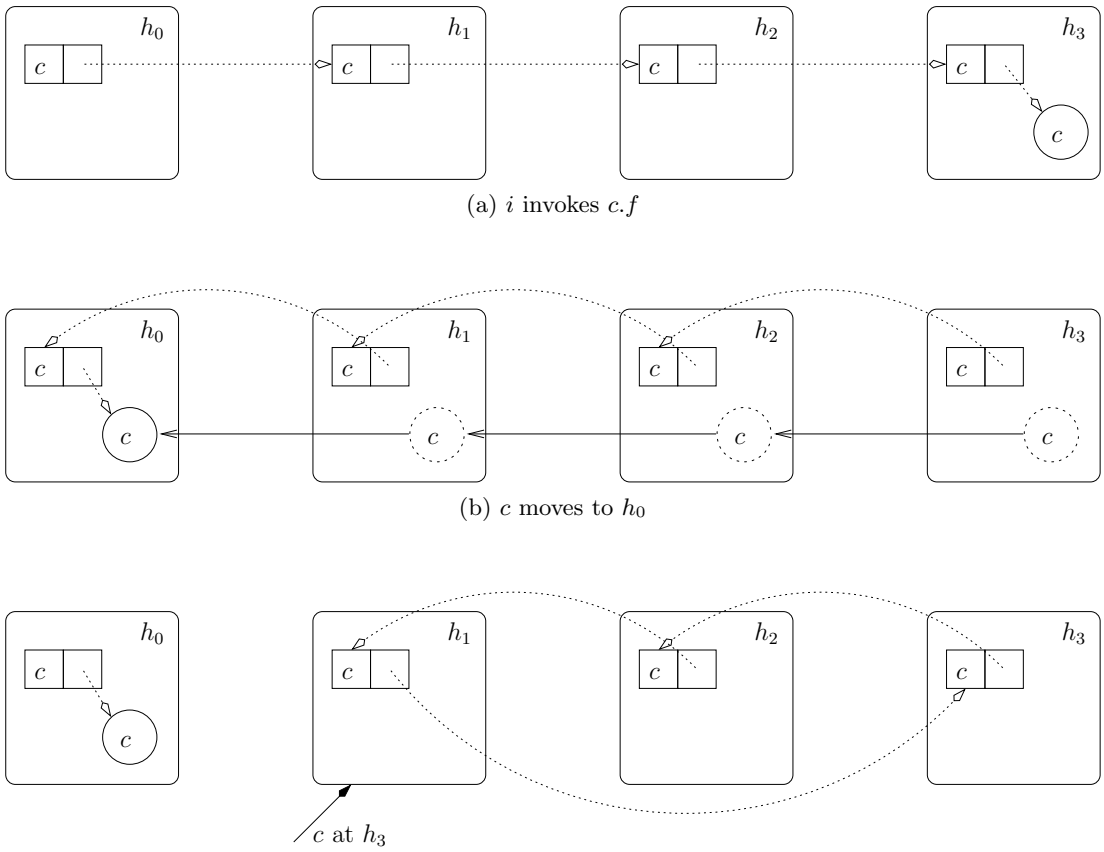
Case $fp(h_d) \neq$ nil: Since any connected subgraph of a directed tree is also a directed tree, the subtree rooted at h_d is a directed tree and so is the subtree rooted at h_s after the subtracting the subtree rooted at h_d. Rule 2 removes the arc from h_d to its parent on the path to h_s in G, overwriting it with a self-arc that we ignore in the definition of G'. Rule 2 also adds a new arc from h_s to h_d. This arc connects the tree rooted at h_d and the remaining tree rooted at h_s. Since the number of nodes is constant, the number of edges is constant, and G' is connected, G' is a directed tree rooted at h_d, the location of r after the move.

□

5.3.1 Correctness of FP$_c$

Figure 5.6 demonstrates that cycles can arise if one naively adds path collapsing to the unmodified FP algorithm. In Figure 5.6a, the finder f running at $h \in H - \{h_0, h_1, h_2, h_3\}$ follows the chain of forwarding pointers to r at h_3 *i.e.* at a host not depicted in Figure 5.6. The finder f then creates messages to update the forwarding pointers along the chain of forwarding pointers it traversed while searching for r. The purpose of these messages is to collapse the chain of forwarding pointers to point directly to h_3. A message is sent to h_0 and to h_1, but not h_2, because its pointer is already current, or h_3, r's current residence. In Figure 5.6b, r moves backwards along the chain of forwarding pointers to h_0, before h_1's collapse message arrives. In Figure 5.6c, the collapse message for h_2 arrives and creates a cycle.

To prevent cycles, h_1 must know to reject c: we must define Lamport's "happens-before" relation for forwarding pointers. As Fowler observed, counting the moves a mobile object makes is sufficient to distinguish which of two forwarding pointers is more recent [30]. However, doing so sacrifices any information contained in a timestamp derived from a



(a) i invokes $c.f$

(b) c moves to h_0

(c) i updates h_1

Figure 5.6: Path Collapse Introduces Forwarding Pointer Cycle

physical clock, such as how old a forwarding pointer is. A Lamport clock requires the pairwise exchange of messages between all nodes interested in totally ordering some event. This can be expensive and unnecessary if one knows some statistical properties of the physical clocks that compose the system. Below, we implement a very simple and cheap clock and, in so doing, allow a location service to retain, but not rely on, physical clocks. A proof using a clock that counts moves is a straightforward simplification of this proof.

Proof Idea: Realizing FP with path collapse (FP_c) requires additional rules. For each search, a finder creates a unique marker, a "bread crumb," with which it marks the hosts it traverses while following the chain of forwarding pointers to a particular resource. Then it sends collapse messages that share the bread crumb marking to each host along the chain. The host only accepts the collapse message if its marking matches the host's current marking.

Let $id : F \to \mathbb{N}$ assign a unique identifier to each finder $f \in F$. Let $count : F \to \mathbb{N}$ assign a unique identifier to each find issued by a particular finder.

Let $B = \mathbb{N} \times \mathbb{N}$ denote the set of "bread crumbs." The first component of a bread crumb either uniquely identifies a finder or denotes the null, or "scrubbed," marking, and ranges from 0 to $|F|$, the cardinality of $F + 1$. The second component sequentially identifies the finds issued by that finder. Let $m : H \times R \to B$ mark the fact that a finder has deposited a bread crumb on h for some r. Let $FM = R \times B$ be the set of find messages. Let $Moves = R \times H \times H$ be the set of resource moves.

Let $CM = \{(h_0, r, h_1, b) \in H \times R \to H \times B\}$, where $h_0 \neq h_1$, denote the set of collapse messages. The collapse message $c \in CM$ is a four tuple consisting of a target host, a resource, a forwarding pointer, and a bread crumb. Note the constraint that the target host and the forwarding pointer must be distinct. Let $H_{CM} = \{h \mid (h, r, h_i, b) \in CM\}$ denote the subset of hosts for which a collapse message is pending.

FP$_c$ Rules: To the FP rules above, we add the following:

1. **Create Find Message**: At the start of a search for r, the finder f creates the find message $f_m = (r, b) \in FM$, where $b = (id(f), count'(f))$, for $count'(f) = count(f) + 1$.

2. **Drop Crumb**: Whenever the host h responds to the find message (r, b) with a forwarding pointer, it sets $m' = m((h, r) := b)$.

3. **Eat Crumb on Move**: In addition to the actions in Rule 2 above, the arrival of r at h sets $m' = m((h, r) := (0, 0))$ to scrub the bread crumb at h.

4. **New Collapse Messages**: Each search issued by each finder adds an element to CM for each host it visits along its path to r, except the host on which the finder is running, the host on which r is found, and the penultimate host.

5. **Receive Collapse Message**: When $h_c \in H_{CM}$ receives $c = (h_o, r, h_1, b) \in CM$, h_c drops c if $h_c \neq h_0$ or if $b \neq m(h, r)$. Otherwise, h_c sets $m' = m((h_c, r) := (0, 0))$ and updates its forwarding pointer: $f' = f((h_c, r) := h_1)$.

Equation 5.13 uses a binary bread crumb to illustrate why each crumb must be unique. Unique crumbs can prevent collapse messages from being processed out of order; however, here we use them only to kill all collapse messages sent prior to a scrubbing.

$$\begin{aligned} i_0 \text{ marks } h &\implies m(h,r) = T \\ r \text{ transits } h &\implies m(h,r) := F \\ i_1 \text{ visits } h &\implies m(h,r) := T \\ i_0 \text{ sends collapse to } h &\implies \text{BOOM} \end{aligned} \tag{5.13}$$

Theorem 5.3.2. *Under FP_c with path scrubbing, $\forall r \in R, \forall h_c \in H_{CM}, G = (H_k, E)$ is a directed tree rooted at h_r, the current location of r, after $n > 0$ tree mutations.*

Proof by Strong Induction.

Base case: This base case is identical to that in Theorem 5.3.1 above.

Inductive step: Assume after n tree mutations, G is a directed tree rooted at h_r.

Under FP_c, three rules that mutate the tree of forwarding pointers, local update, the receipt of a collapse message and a move.

Case Local Update: See this case in Theorem 5.3.1 above: the effect of the collapse messages that occur prior to this local update do not violate the tree invariant because of the inductive hypothesis.

Case Move: See this case in Theorem 5.3.1 above: the effect of the collapse messages prior to this move do not violate the tree invariant because of the inductive hypothesis.

Case Collapse Message: Let receipt of the collapse message $c = (h_c, h_d, b) \in CM$ be the $n + 1$ tree mutation. There are two cases:

Case $m(h_c, r) \neq b$**:** Another find overwrote b at h_c with its bread crumb, or r transited h_c and scrubbed b. In either case, h_c drops c and G is a tree by the inductive hypothesis.

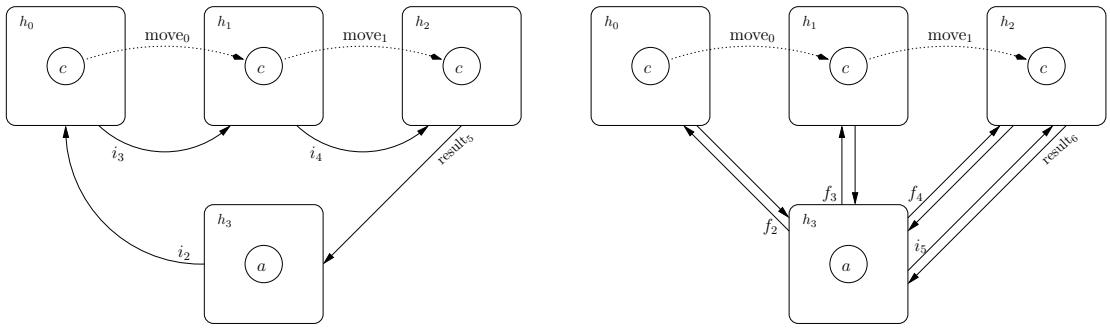
Case $m(h_c, r) = b$**:** At the time that b was dropped, h_c was the root of a direct subtree and h_d was the root of a directed subtree at that same time, by the strong inductive hypothesis. Any moves r made after b was dropped also created trees, by the strong inductive hypothesis. Thus, even if r moved into h_c's subtree or h_d subtree, thereby moving a host formerly in those subtrees to the root, the h_c and h_d subtrees remained trees. At the time of the receipt of c, the h_d's subtree, with the h_c subtree removed, is still a tree. Thus, when we act on c, we form G' by making the h_c subtree a direct subtree of h_d. We conserve edges and connectivity. Therefore, G' is still a directed subtree.

□

5.4 A Comparison of Two Invocation Protocols under FP

Under a location service that uses a central directory, a natural invocation protocol for mobile components is to first find the target mobile component, then invoke an operation on it. We call this protocol "find, then invoke" (FI). This protocol generalizes easily to forwarding pointers — the invoker simply issues finds to follow the chain of forwarding pointers.

A location service based on forwarding pointers also makes feasible another invocation protocol that integrates name resolution and routing — self-routing invocations (SI). Under SI, when a host receives an invocation message for an absent component, it forwards that invocation to the next host in the chain of forwarding pointers to that component. The SI protocol does not make much sense in the absence of forwarding pointers, because each host that receives an invocation that fails would be burdened with querying the directory on the invoker's behalf to forward the invocation.



(a) Self-Routing Invocations (b) Find, then Invoke

Figure 5.7: Two Invocation Protocols

Intuitively, SI is appealing because it sends fewer, albeit larger, messages and, as we show below, suffers from a narrower race window than does FI. As this section unfolds, we show that, to the contrary, FI is superior except in the pathological case of the target mobile object moving at very nearly the rate at which invocations are generated. The decisive difference between the two protocols is message size. This chapter shows that successfully integrating name resolution and routing depends on the ratio of the size of a find message to the size of the message routed to target. This ratio is an important, and heretofore neglected, consideration for any project that proposes protocols that integrate name resolution and routing, such as intentional naming [3] and location independent invocation [14].

In Figure 5.7, f is a find message, i is an invocation message, and a seeks to invoke an operation on c. In both figures, the last known location of c at h_3 is h_0. Before a's invocation reaches it, c moves twice, reaching h_2. Each message is subscripted with its temporal order. In SI, an invocation itself follows the chain of forwarding pointers to the component for which it was launched, as shown in Figure 5.7a. Figure 5.7b depicts an invocation under FI. Here, the invoker a sends find messages to follow the chain of forwarding pointers for c to h_2, where it sends the invocation.

Under MAGE, the collocation of an invocation message and its target mobile component at a particular host does not necessarily imply execution at that host, as a mobility attribute is unlikely to have specified that host as the execution target. Thus, for both SI and FI, a MAGE invocation usually entails an additional hop to its execution target, where the

host at which the invocation caught up to its target mobile object forwards the invocation message, swollen with the mobile object. This additional work adds the constant cost of a single additional invocation message to both protocols, which we do not include in the analysis that follows.

The two protocols trade-off the number of messages for their size. For $x \in \{f_q, f_r, i\}$, let $|x|$ be the size of the message. The message f_q is a find query, and f_r, a find reply. For brevity and because we usually work with the find messages as a unit, we define $|f| = |f_q| + |f_r|$. Almost always, $|f| << |i|$ holds, since a find query contains only the name of its target mobile component and a find reply contains only a location, while an invocation message minimally contains not only the name of its target mobile component, but also a target method name, not to mention that method's parameters, each one of which may be quite large.

Intuitively, it would seem that this observation about messages sizes closes the case: FI is superior to SI. However, while both protocols are subject to races with their target mobile component, FI's race window has twice the duration of SI's since FI must both send f_q and wait for f_r, its reply.

As in Section 5.2, we use M to denote the moves a mobile component makes and V the invocations an invoker makes, both over the same time interval. M and V are Poisson distributed random variables. Recall that r is the rate at which an invoker sends messages and that we assume $\lambda_m < r \leq \frac{1}{t_r}$, for $t_r = \text{RTT}$. The analysis that follows assumes a single invoker, abstracting other invokers into the moves made by the target component.

Consider Figure 5.7b again. What if c moves before the invocation arrives at h_2? After FI's find phase ends, a race occurs between c leaving h_2 for some other host h_4 and the arrival of the invocation at h_2. To handle this race, FI restarts, and begins a new find phase.

Figure 5.8 depicts the FI race in a sequence diagram. The actor a's find request reaches h_2 while c is still on h_2. Once h_2 replies saying it hosts c, the race starts. There are two races, both involving r_1. If r_1 occurs before r_0, then a will know to continue following the chain of forwarding pointers. This case is indistinguishable from the case where c was simply not at h_2 when a's find arrived. The second race is between r_1 and r_2; the race window is

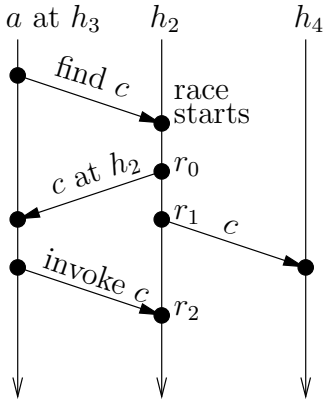


Figure 5.8: The FI Race

lower-bounded by RTT, the time for h_2 to send its reply to a's find and a dispatch of its invocation message to arrive at r_2. If r_2 wins the race, then c executes the invocation and the FI protocol ends; if not, the chase continues. In Figure 5.8, r_1 wins the race.

The probability that FI wins the latter race with its target mobile component is $e^{-\lambda_m t_r}$, for $t_r = \text{RTT}$, because this is the probability of zero moves after a successful find reply is sent and the invocation arrives.

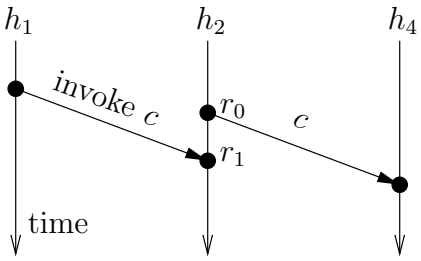


Figure 5.9: SI Race

SI too is subject to a race. Figure 5.9 starts in the configuration that Figure 5.7a depicts. Here, r_0, c's departure from h_2 for h_4, races against r_1, i's arrival at h_2. Ignoring i's size, i's travel time, and thus the race window, has duration $\frac{t_r}{2}$, since neither the invoker nor the invocation forwarder need to wait for replies.

The probability that SI wins its race with its target mobile component is $e^{-\lambda_m \frac{t_r}{2}}$, because this is the probability of 0 moves after an invocation message has arrived at h_0 and before it arrives at h_1.

Next, we compare these two invocation protocols for forwarding-pointer-based directory services and show that, in the expected case, FI uses less bandwidth than SI, in spite of its longer race window.

The moves the target mobile component made before the beginning of an invocation must first be consumed. Figure 5.10 defines t_b, the time between the end of one invocation protocol and the start of another. We do not consider the interval between the start of the two invocation protocols because, at the time of the successful invocation, the invoker, by definition, knew the location of the target mobile component. Ignoring the possible collocation of invoker and its target component, $t_b > \frac{1}{r}$, inverse of the rate at which the invoker can send messages. Like λ_m, t_b is a parameter of the model.

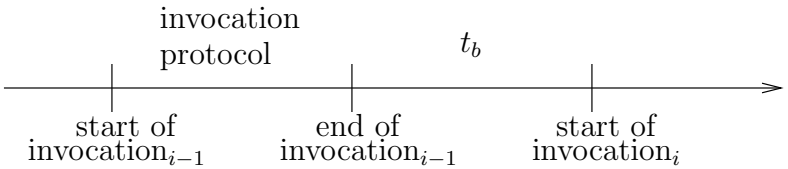


Figure 5.10: Invocation End to Start Interval

The expected number of moves $\langle M \rangle$ in t_b is $\lambda_m t_b$. Traversing the chain of forwarding pointers formed by these moves takes time, during which the mobile component can again move, and so on. The time to traverse the forwarding pointers formed while traversing the path formed during t_b is $(\lambda_m t_b)t_r$, so the expected number of moves that occur during this time is $\lambda_m(\lambda_m t_b)t_r$. In turn, the time to traverse the path of forwarding pointers formed by the moves made while traversing these moves is $(\lambda_m(\lambda_m t_b)t_r)t_r$, which generates $\lambda_m(\lambda_m(\lambda_m t_b)t_r)t_r$ expected moves.

Let U be the number of moves until the invoker first catches up to its target component.

$$
\begin{aligned}
\langle U \rangle &= \lambda_m t_b + \lambda_m(\lambda_m t_b)t_r + \lambda_m(\lambda_m(\lambda_m t_b)t_r)t_r + \cdots \\
&= \lambda_m t_b(1 + \lambda_m t_r + \lambda_m^2 t_r^2 + \cdots \\
&= \lambda_m t_b \sum_{j=0}^{\infty} (\lambda_m t_r)^i \\
&= \lambda_m t_b \frac{1}{1 - \lambda_m t_r} && \lambda_m < \frac{1}{t_r} \\
&= \frac{\lambda_m t_b}{1 - \lambda_m t_r} && (5.14)
\end{aligned}
$$

5.4.1 FI Cost Analysis

Under FI, an invoker has *caught up* with its target mobile component c, when it receives a reply to a find message that states "c is here." FI alternates between sending finds to catch up to its target mobile component, and sending invocations. After each failed invocation, it must again catch up. As a simple optimization, the reply to a failed invocation contains the forwarding pointer, which FI can immediately follow upon restarting the find phase of its protocol. Equation 5.14, for $t_r = \text{RTT}$, is the message cost of FI initially catching up to c, after the start of a new invocation protocol.

The time it takes the invoker to again catch up with the mobile component is Equation 5.14, but with t_b replaced by $\lambda_m t_r$ multiplied by RTT, the time to traverse the forwarding pointer chain created by any moves that occurred during the transit time of the reply to the successful find and that of the failed invocation message. Since $\lambda_m t_r \text{RTT} = \lambda_m t_r^2$, the expected number of messages to catch up again, after each failed invocation, is

$$
\frac{\lambda_m \lambda_m t_r^2}{1 - \lambda_m t_r} = \frac{(\lambda_m t_r)^2}{1 - \lambda_m t_r} \tag{5.15}
$$

$$\begin{aligned}\langle \mathrm{FI}_{ac} \rangle &= |i| + P(j > 0 \text{ moves before call arrives})[|f|(E(\text{finds}) \text{ after a failed call}) \\ &\quad + |i| + P(j > 0 \text{ moves before call arrives})(|f|(E(\text{finds}) \text{ after a failed call}) + \cdots)] \\ &= |i| + (1 - e^{-\lambda_m t_r})[|f|\frac{(\lambda_m t_r)^2}{1 - \lambda_m t_r} + |i| + (1 - e^{-\lambda_m t_r})(|f|\frac{(\lambda_m t_r)^2}{1 - \lambda_m t_r} + \cdots)] \end{aligned} \tag{5.16}$$

In Equation 5.16, $|i|$ is the cost of an invocation (call) and $|f|$ is the cost of a find message. After a successful find, the chance that a message, in particular an invocation, reaches the mobile component's host before it moves is $P(\text{no moves in } t_r) = e^{-\lambda_m t_r}$, from the definition of Point Poisson. Thus, after each find phase, the chance that the invocation we dispatch wins the race with the mobile component and arrives in time is $e^{-\lambda_m t_r}$ and the chance it does not is $P(j > 0 \text{ moves before call arrives}) = 1 - e^{-\lambda_m t_r}$.

Each time the find phase of FI catches up with the mobile component, it sends an invocation message which either succeeds or fails. An invocation fails whenever the target mobile component moves after the reply to a successful find is sent from a host and while the invocation is in transit to that host. If it fails, the invoker must first catch up again and then send another invocation message. $\frac{(\lambda_m t_r)^2}{1-\lambda_m t_r}$ is the $E(\text{finds})$ after a failed call from Equation 5.15.

$$\begin{aligned} a &= |f|\frac{(\lambda_m t_r)^2}{1 - \lambda_m t_r} + |i| \\ b &= 1 - e^{-\lambda_m t_r} \end{aligned} \tag{5.17}$$

Next, we derive a closed form solution of Equation 5.16. We form Equation 5.18 from Equation 5.16, using a and b, as defined in Equation 5.17. We then rearrange Equation 5.18 to make clear that it contains a geometric series.

$$\begin{aligned}
\langle \mathrm{FI}_{ac} \rangle &= |i| + (1 - e^{-\lambda_m t_r})[|f|\frac{(\lambda_m t_r)^2}{1 - \lambda_m t_r} + |i| + (1 - e^{-\lambda_m t_r})[\cdots]] \\
&= |i| + (1 - e^{-\lambda_m t_r})[(a+b)[(a+b)[(a+b)(\cdots]]] \\
&= |i| + (1 - e^{-\lambda_m t_r})a(1 + b + b^2 + \cdots + b^n) \\
&= |i| + (1 - e^{-\lambda_m t_r})a\sum_{i=0}^{\infty} b^i \\
&= |i| + (1 - e^{-\lambda_m t_r})\frac{a}{1-b}
\end{aligned} \tag{5.18}$$

Finally, we undo the rewriting and simplify. Equation 5.19 captures the expected total data, and implicitly the expected messages, FI transmits, after the invoker has first caught up with its target mobile component and dispatched an invocation message.

$$\begin{aligned}
\langle \mathrm{FI}_{ac} \rangle &= |i| + (1 - e^{-\lambda_m t_r})\frac{|f|\frac{(\lambda_m t_r)^2}{1-\lambda_m t_r} + |i|}{1 - (1 - e^{-\lambda_m t_r})} \\
&= |i| + (1 - e^{-\lambda_m t_r})\frac{|f|\frac{(\lambda_m t_r)^2}{1-\lambda_m t_r} + |i|}{e^{-\lambda_m t_r}} \\
&= |i| + (e^{\lambda_m t_r} - 1)(|f|\frac{(\lambda_m t_r)^2}{1 - \lambda_m t_r} + |i|)
\end{aligned} \tag{5.19}$$

For $|i_r| = |i| + |f_r|$, D_{fi} is the data sent by FI, until the invocation i and its target component are collocated.

$$\begin{aligned}
\langle D_{fi} \rangle &= E(\text{finds}) \text{ to first catching up} && + E(\text{messages}) \text{ after first catching up} \\
&= |f|\frac{\lambda_m t_b}{1 - \lambda_m t_r} && + |i| + (e^{\lambda_m t_r} - 1)(|f|\frac{(\lambda_m t_r)^2}{1 - \lambda_m t_r} + |i_r|)
\end{aligned} \tag{5.20}$$

We use $|i_r|$ to denote the size of invocation message paired with a find reply, the optimization under which a failed invocation acts as a find query.

5.4.2 SI Cost Analysis

The expected data transfer required (and implicitly the number of invocation messages) used under SI, $\langle D_{si} \rangle$, differs from $\langle D_{fi} \rangle$ in two ways: We replace

1. t_r with $\frac{t_r}{2}$, because, under SI, an invocation is a find, so at each step on the chain of forwarding pointers, we do not notify the invoker, but rather simply dispatch the invocation; and

2. $|f|$ with $|i|$.

$$\begin{aligned} \langle D_{si} \rangle &= E(\text{calls) to first catching up} && + E(\text{calls) after first catching up} \\ &= |i| \frac{\lambda_m t_b}{1 - \lambda_m \frac{t_r}{2}} && + |i| + (e^{\lambda_m \frac{t_r}{2}} - 1)(|i| \frac{(\lambda_m \frac{t_r}{2})^2}{1 - \lambda_m \frac{t_r}{2}} + |i|) \end{aligned} \quad (5.21)$$

5.4.3 FI vs. SI

In these figures, we use RTT $= t_r = .606$ms. The UNIX utility `ping` run for ten minutes in February 2008 on the switched 100Mb Ethernet LAN in our laboratory reported this number as the average RTT. Unless otherwise noted, we fix the remaining variables in Equations (5.20) and (5.21) to following values, which we believe are reasonable to assume: $|i| = 5|f|, \lambda_m = \frac{1}{t_r}, t_b = 10t_r = 6.06$ms, and $|f_r| = .5|f|$ so $|i_r| = |i| + .5|f|$. The dependent variable is the data sent, denominated in multiples of $|f|$, until an invocation message is collocated with its target component.

Figure 5.11 varies $|i|$ as a multiple of $|f|$. Figure 5.12 varies λ_m as a fraction of RTT. Figure 5.13 varies t_b in multiples of RTT. Except in in Figure 5.12, $\langle D_{si} \rangle > \langle D_{fi} \rangle$, and, even here, SI only sends less data when a component moves often relative to RTT. Thus, we conclude that FI is the superior invocation protocol, in spite of its wider data race window.

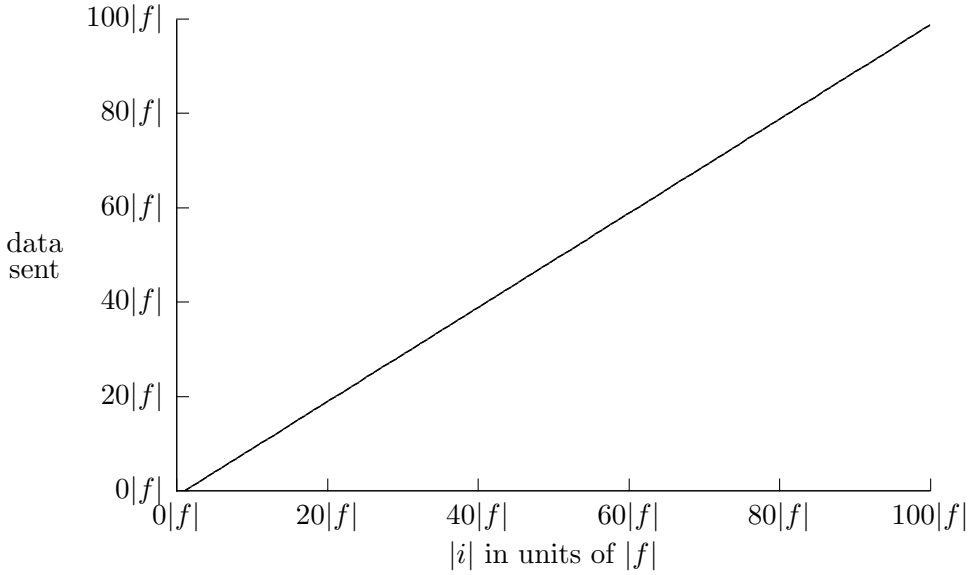


Figure 5.11: Data Sent Delta

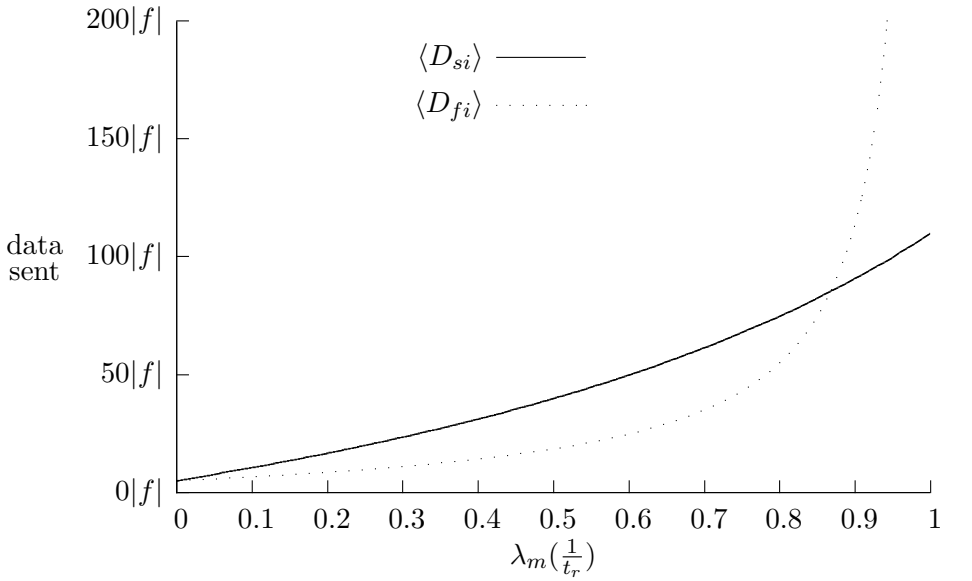


Figure 5.12: λ_m as a Percentage of RTTms^{-1}

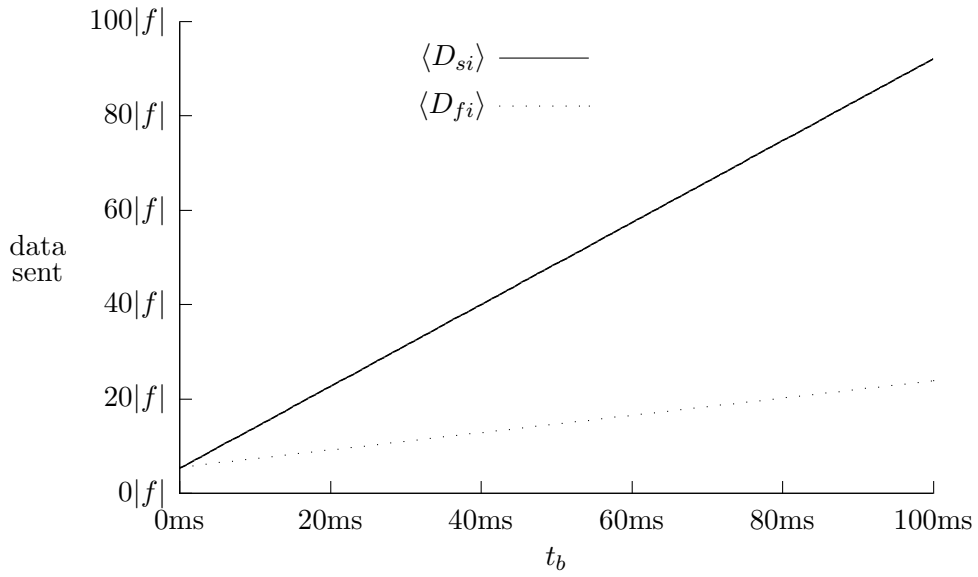


Figure 5.13: Varying t_b

5.5 Related Work

Forwarding pointers are not new. They were first used in the DEMOS/MP distributed system [83], which called them links and updated them only when a process moved to a machine that had links pointing at that process. They were widely used in other early mobile objects systems [14, 48, 96]. A number of distributed garbage collection techniques use forwarding pointers [71, 81]. Researchers have also proposed and evaluated location services that incorporate forwarding pointers for Personal Communication Services (PCS), the infrastructure cellphones use [18, 19, 59].

In general, these projects describe the design and implementation of location services based on forwarding pointers, but do not prove their correctness or analyze their cost. Below, we review the work that, like this chapter, does tackle these questions.

5.5.1 Expected Message Cost and the Correctness of FP

Section 5.2 presents concurrent models of three location services in the presence of mobility — a directory and two variants of forwarding pointers. Thus, our expected cost analysis handles (1) multiple invokers; and (2) mobile objects moving while lookups and invocations

occur. As a result, our analysis incorporates data races between invocations and moves. We report bounds in terms of messages. Section 5.3.1 proves the correctness of a *concurrent* path-collapsing forwarding pointers protocol.

Fowler's "The complexity of using forwarding addresses for decentralized object finding" is the seminal analytic work on forwarding pointers [30]. Fowler was the first to observe that incrementing a counter in a mobile object each time it moves is a sufficient timestamp for a forwarding pointer since we only need to know which of the two pointers is more recent. Fowler formulates and analyzes three forwarding pointer variants, which he calls Lacc, Jacc, and PCacc. Lacc is the naïve forwarding pointer protocol with no updates; Jacc includes redirecting the local forwarding pointer after a successful find; and PCacc collapses the path. He presents lower and upper bounds for these variants.

Fowler considers only forwarding pointers. He does not consider integrating name resolution and routing. Like us (Section 5.2), Fowler assumes that machines do not fail, so he does not consider fault tolerance and FP's fault sensitivity. Fowler does not prove the correctness of his forwarding pointer protocols. Unlike our analysis, Fowler counts access (find) attempts; that is, the injections of an access message. He does not count the re-sends of that message during the traversal of a chain of forwarding pointers. Thus, Fowler's model does not account for races between moves and accesses. Fowler's PCacc does not handle concurrent accessors (invokers), as our FP_c does. In short, he analyzes a simpler and less practical problem than we have in this chapter.

Shapiro *et al.* proposed a form of forwarding pointers with a focus on garbage collection [90, 91]. They gave their proposal the sobriquet "SSP Chains," where SSP stands for stub scion pairs. Scions are skeletons and are used during local garbage collection to reach and keep alive stubs. A stub has fixed targets, usually scions, and does not contain a universal unique identifier (UUID) for its target mobile object; Shapiro *et al.* claim removing UUIDs from stubs is essential for performance. Sending a remote reference or migrating an object extends SSP chains. The former complicates the use and maintenance of SSP chains and is a direct consequence of the fact that SSP stubs do not contain a UUID and therefore cannot query a name service to follow the chain to their target object. In this

thesis, a remote reference is a stub that *does* contain a UUID and can therefore directly query the name service to traverse the FP chain to its object. Practically, this is why the FP variants proposed in this thesis do not extend an FP chain when a reference moves from one host to another.

Shapiro *et al.* never shorten an SSP chain, except to reclaim the chain when it has no users. A shortcut is the update of the local forwarding pointer after a successful lookup. In SSP, a shortcut is only a hint, stored as a *weak* reference. Finds do not alter *strong* SSP chain, which is created by object and reference movement creates. These properties allow SSP chains to support concurrent access. They suggest path-collapse as a possible optimization, but, like a shortcut, only as a hint. In contrast, this thesis proposes, analyzes, and proves correct FP variants that *do* alter shared forwarding pointer chains.

Shapiro *et al.* consider machine failure although they rely on the strong assumption of a failure detection oracle. We have left machine failure for future work. They do not provide an asymptotic analysis of their protocols, nor prove their correctness.

The Arrow Distributed Directory Protocol is a simple protocol that acquires exclusive access to a mobile object [22]. When a node issues a find for a target mobile object, each arc in the forwarding pointer tree that the message traverses on its path from the finder to the object flips to point toward the finder. When the find messages reaches the object, a new tree rooted at the finding node forms. The object then traverses the newly formed path to the finder, which then has exclusive access to the mobile object. The arc-flipping caused by a find message traversal disconnects the tree while the find message is progressing toward its target. This fact handles contention elegantly. An invoker in the finder's subtree will find the finder and block there waiting for the finder to release the object. An invoker in the object's subtree proceeds normally. Demmer *et al.* prove the correctness and complexity bounds of the Arrow protocol.

The Arrow protocol only handles exclusive access: in it, the accessed mobile object always moves to the finder's location. In terms of MAGE, this is equivalent to an invoker using only COD. The protocols analyzed in this chapter allow executing operations in place, as well as moving the target object to an arbitrary node in the graph. The Arrow protocol

is well-suited for use in realizing write access in distributed shared memory, as a review of its citations in the research literature makes clear.

Moreau formalizes a fault tolerant forwarding pointer location service [72, 73]. He describes a mechanical proof of its correctness using the proof assistant Coq [12]. As we have noted above, naïve forwarding pointer protocols are fault sensitive: the probability that an object becomes unreachable increases with the length of a forwarding pointer chain. Moreau's protocol addresses this problem by collapsing a length n suffix of the forwarding pointer chain. A mobile agent remembers the last n hosts it visited. Upon arrival to a new host, it updates the forwarding pointers of those n hosts.

Moreau's protocol enqueues messages at a mobile object's last host during migration rather than tolerating transient cycles (as we do), which works well when you include a laptop as a possible host. After arriving at its new host, the mobile object itself collapses the path and is unavailable until it finishes. Multiple invokers simply queue at the mobile object's last location. Their finds can race with the suffix collapse, but if they lose, they simply follow a longer path to the object. Thus, Moreau's protocol handles concurrent invokers. Recall that we named our the path-collapsing FP variant FP_c. The price FP_c pays asynchronously after an invocation, Moreau pays during migration, since the suffix collapse occurs during migration. Many invokers cooperatively drive the FP_c cost to zero; Moreau's cost is fixed at n per migration.

Moreau does not analyze the bounds of his protocols, except when he considers the cost of collapsing the suffix of an FP chain. Thus, he does not confront the difficulty of accounting for races between moves and finds. Relying on 25,000 Coq tactics[3], Moreau's correctness proofs are much longer and more complex than the ones we have presented in this chapter.

[3] A deduction rule links premises and a conclusion. In backward reasoning, the conclusion is the goal and the premises are subgoals. Tactics implement backward reasoning. When applied to a goal, a tactic replaces the goal with its subgoals. Thus, a Coq tactic is a command that realizes one or more steps of a proof [98, Chapter 8].

5.5.2 Directory vs. Forwarding Pointers

Section 5.2.4 compares the lookup and maintenance cost of a single directory to two forwarding pointers variants using expected analysis that incorporates races between invocations and moves.

In a distributed setting with mobile objects, like MAGE, Alout *et al.* use Markov models to compare the access (find) cost of FP and D [5]. Their model is quite detailed: for instance, it uses random variable to model such things like migration time, where our model implicitly uses expected time via RTT. An important difference is that they consider only access time, not the time to update the directories. Further, their analysis assumes a mobile object does not revisit a host. They assume that chains of forwarding pointers are not shared, that each accessor/object pair has its own chain. Under this assumption, path collapse does not make sense. However, they do not consider shortcuts, which are relevant to and could improve the performance of their protocol. Like us, they assume hosts do not fail.

Finding an optimal location service for Personal Communication Services (PCS) is important to the telecommunications industry, because it would save money in infrastructure costs and improve service. Thus, a number of researchers have compared forwarding pointers to a directory, which is called a home-location register (HLR) in the PCS literature,

Jain *et al.* attack precisely the question we have asked: under what circumstances is FP superior to D [47]. Their callers do not shortcut, *i.e.* cache the result of a find. They do not consider path collapse. They bound the length of forwarding pointers: every k move, they update the HLR. In the PCS context, a FP protocol must send a message back to the previous visitor-location register (VLR), which is analogous to a host in our analysis, because a mobile device can only detect that it has left that VLR after the fact. These differences lead them to show, under various probabilistic assumptions about a mobile user's movement behavior, that FP only wins when the call to move ratio is less than 0.5, in contrast to our result which is independent of the call to move ratio.

Krishna addresses the D vs. FP question in his dissertation [54, Chapter 3]. He models the problem using time-cost, not messages. He considers two variants of bounded-chain FP,

one which, like the variant of Jain *et al.*, updates the HLR every k moves and another which updates the HLR after some k searches. Finally, he introduces shortcutting, which he calls "search-update," versions of these two variants. Like Jain *et al.*, he shows that FP performs better when the call to move ratio is low.

Chang *et al.* propose a hybrid location service that uses forwarding pointers to build a virtual local directory out of adjacent VLRs that a user frequently visits [18]. Only when a user moves out of the virtual directory does the HLR need to be updated. To compare their scheme to competing alternatives, including directory and FP, they count the update messages sent to the HLR and VLRs during a single itinerary. Under this itinerary, the FP variant they consider sends 1 message to the HLR and 9 to VLRs, the directory approach sends 5 messages to the HLR and 5 to VLRs, while their approach sends 1 message to the HLR and 6 to VLRs. Unsurprisingly, their approach performs best under the itinerary they choose. The FP variant they consider does not shortcut or collapse paths.

5.5.3 FI vs. SI

Section 5.4 compares two invocation protocols based on forwarding pointers — one that maintains the traditional separation of name resolution and routing (FI) and one that integrates them (SI). We show that message size is a critical factor in determining which to use, and that integration does not make sense when the routed message is larger than a find message, as generally holds when the routed message is an invocation. We demonstrate this result holds when the routed message is 5 times larger than an find message.

The Hermes project integrates name resolution and routing in its invocation protocol [14]. "Intentional naming" [3] proposes a location service that integrates name resolution and message routing. Neither of these projects consider the conditions under which integrating name resolution and message routing makes sense. To our knowledge, we are the first to consider this question.

5.6 Future Work

This chapter focuses on two directory schemes — a single, centralized directory and forwarding pointers. An interesting avenue for further research is to analyze other directory schemes in terms of their message cost. For instance, a D-FP hybrid that partitions the hosts among a set of directories. Under this scheme, when a mobile component leaves one directory's region, it forms a chain of forwarding pointers that connect each region's directory. It would also be interesting to extend this analysis to the distributed hash tables, such as Chord [94], used in P2P settings. We showed that FP is superior to D for a single invoker in Section 5.2.4. We intend to extend this analysis to multiple invokers for FP and FP_c.

The random model of movement and invocation presented here uses Poisson random variables. With the aid of application specific knowledge, such as traces, it may be possible to build more accurate models. For presentation clarity, the analysis presented in this chapter assumed that machines do not fail. We plan to revisit our analyses in the presence of machine failure and explore fault tolerant versions of the protocols we have considered, in addition to quantifying the fault tolerance protection afforded by frequent path collapse. Finally, one could use simulation and experiment to further explore this problem space and verify the analysis.

5.7 Summary

In this chapter, we have discussed location services in the presence of mobility. We presented forwarding pointers and contrasted them with a single centralized directory, and showed that forwarding pointers require fewer find messages on average. We defined and proved correct a concurrent path-collapsing variant of forwarding pointers. We then introduced and analyzed two invocation protocols built on forwarding pointers — "find, then invoke," which maintains the traditional separation of name resolution and routing, and "self-routing invocations," which integrates the two. Conservatively assuming invocations are larger than finds, we show that the "find, then invoke" protocol requires a smaller expected data transfer before a

successful invocation than “self-routing invocations” except when a mobile component moves rapidly, which given the cost of movement, is unlikely to persist in well-behaved applications.

Chapter 6

Implementation

Program construction consists of a sequence of refinement steps.

Niklaus Wirth

In this chapter, we discuss the challenges faced in the implementation of MAGE. First, we set the stage with a brief description of RMI's runtime, against which we present the MAGE runtime as a sequence of modifications and extensions. Then, we broadly follow the outline of Chapter 4, and describe the challenges in implementing the MAGE primitives and operations on those primitives. In closing, we discuss the limitations of the current implementation, then summarize the chapter.

6.1 Challenges

In principal, MAGE could be built on any language that provides mobility, such as Java, D'Agents [36], Messengers [31], or Ajanta [102]. We choose Java as our implementation platform because of its platform independence, widespread availability, and support for remove calls via RMI and serialization, hereafter referred to as marshaling[1]. We realized

[1]So far as the author knows, Sun introduced the use of the term serialization, as opposed to marshaling, to refer to the writing of data, such as an object, to a bit string. In fact, Sun distinguishes serialization and marshaling, which it restricts to codebase-annotated serialization [87]. This distinction is highly Java-centric. In spite of Sun's marketing prowess and the success of Java, this author prefers "marshaling" because serialization already has a useful technical definition in concurrent programming where it refers to the

MAGE as a Java class library to ease and facilitate its deployment. The MAGE runtime system is built upon and extends the Java RMI class library and its runtime system.

The RMI's registry was inadequate as it is designed to catalog `Remote` objects on a single host, not a cluster. MAGE uses forwarding pointers, to decentralize MAGE's directory service and for efficiency, as described in Chapter 5. This decision means that every MAGE host must store forwarding pointers for each mobile object that either visits or is invoked from that host. The principal challenge in implementing a forwarding pointer directory service is preventing the formation of cycles. A simpler solution would have been to use a single, centralized directory.

The decision to use Java means that MAGE inherited Java's assumption that objects are immobile, that they spend their entire lifespan in a single address space. This assumption means that

1. the client leases that Java's Distributed Garbage Collector (DGC) server hands out have the remote server immutably embedded within them;
2. a `Remote` object can have static fields;
3. a `Remote` object can always be replaced with a proxy during marshaling, as when it is an actual in a remote invocation; and
4. an invocation, absent synchronization, can always proceed.

To solve the DGC challenge, MAGE modifies Java's DGC to follow an object's chain of forwarding pointers to renew its lease, just as a MAGE invocation must. From the point of view of MAGE applications, DGC occurs out-of-band. Although DGC could, in principle, compete with applications for the CPU and bandwidth, preliminary tests showed no measurable performance impact. MAGE relies on programmers to eschew static fields when defining mobile classes. To prevent RMI always replacing a mobile object with a proxy, MAGE extended the marshaling framework to distinguish whether or not to marshal the

execution of a critical section by one thread at a time.

mobile itself or a proxy to it. The cost of this solution is single condition on a reflective call to determine whether a marshaled object is a descendant of `MageMobileObject`.

Mobility complicates a remote invocation protocol, not so much because a mobile object may have moved once an invocation arrives, but because the mobile object may be preparing to move. When the mobile object has already moved, the protocol can simply be restarted. To move, a mobile object must capture all updates to its state made prior to departure. To capture these updates, MAGE prevents new threads from entering a moving object and waits for threads already running in that object to finish. MAGE sends exceptions to the invokers of post-moving invocations to block the entry of new threads. These invokers immediately restart their invocation protocol, subject to a starvation limit. Draining can, of course, take a long time and fending off invokers that immediately restart is wasteful. Blocking the post-moving invocations in a lock queue and then freeing them all at once, when the executing threads had drained and the mobile object had departed, is likely to lead to better performance. Locking issues surrounding handling movement in the presence of asynchronous incoming calls are tricky, and discussed in detail in Section 6.5.4 which contains Algorithm 6.4.

Mobility attributes further complicate the invocation protocol. The remote invoker must run his local mobility attribute, and embed the result in the invocation. Then the mobile object's host must run the server, or component mobility attribute, if one is bound, and combine that result with the result of the invokers.

When an invocation reaches a mobile object at a host other than the ultimate execution target, the mobile object must move. The connection over which the invoker sent the invocation and, under RMI, would expect to receive the result is to the host at which the mobile object was found. Two solutions leap to mind: 1) The invocation can either be forwarded to the execution target and the result routed back to the invoker via a listener; or 2) the mobile object can move, be locked in place, and the invoker instructed to re-issue its invocation to the execution target. The current implementation chooses the former solution, because no other invocations can occur between arrival at the execution target and the invocation that triggered the move.

> "There are known knowns. There are things we know that we know. There are known unknowns. That is to say, there are things that we now know we don't know. But there are also unknown unknowns. There are things we do not know we don't know."
>
> Donald Rumsfeld
>
> Defense Department Briefing, February 12, 2002

There are undoubtedly many ways to improve the MAGE implementation, many of which are unknown unknowns. Two known known and one known unknown suggestions for improvement include the following:

1. On the hot path of a MAGE invocation, `java.util.HashSet` is instantiated twice. Since the set of hosts in a MAGE cluster is likely to be relatively static, using bit maps to represent sets is likely to improve invocation performance.

2. Invocations on a mobile object made by an invoker collocated with that mobile object are *not* local, but remote call and are therefore marshaled and traverse TCP/IP stack over the loopback device. Each MAGE host could maintain an id → proxy map. Then, when a mobile object arrives, MAGE could use the arriving object's id to set a direct reference to itself in each of its local proxies. Thereafter, calls on these proxies would be local. When id departs, MAGE would null the direct reference in each of id's proxies.

3. Verify the design decision to use a listener to manage results, by empirically comparing the two result management protocols. The fact that the listener allows MAGE to provide fine-grained concurrency via futures[2] is ancillary.

6.2 The RMI Runtime System

Definition 6.2.1. An RMI *remote object* is an instance of `RemoteObject` that can receive and execute remote invocations.

[2] Section 6.3.5 defines and discusses futures.

Service	Map
RMI registry	name → proxy
invocation server	id → `RemoteObject`
DGC server	id → `LeaseInfo`
activation server	id → `RemoteObject`

Table 6.1: RMI Maps

Definition 6.2.2. An RMI *proxy* or *stub* is created for a specific remote object. It stores its remote object's host, or server. It marshals an invocation, sends the marshaled invocation to its remote object's server, and unmarshals the result.

RMI adds two primitives to Java — remote objects and proxies to remote objects. A remote object receives and executes calls that an invoker sends to it via a proxy. To realize this functionality, RMI runs an invocation server, a registry, a distributed garbage collector, and an activation server[3]. Table 6.1 lists these servers and the mappings they maintain. Chapter 2 gives an extended example of an RMI application that illustrates RMI's programming model.

6.2.1 Invocation Server

RMI's invocation server[4] listens for invocation messages, looks up each invocation's target remote object, executes the invocation in the context of that remote object, and replies with the result. An `ObjID` is a class whose instances are globally unique identifiers for remote objects. The invocation server maintains a map of identifiers to remote objects. Proxies embed their remote object's identifier in each call they marshal. RMI's invocation server extracts this identifier to look up the target remote object.

Definition 6.2.3. *Exporting* is the act of binding an identifier to a RMI remote object with an invocation server so that it can receive and execute incoming, remote calls.

[3]RMI's activation server allows a server to export remote objects that hibernate (are stored on disk not in memory) unless a client invokes them. MAGE does not support the hibernation of its mobile objects.

[4]The invocation server is called the RMI server in the Java RMI documentation. For clarity, we use the more precise name.

6.2.2 Distributed Garbage Collection

When you couple a distributed programming model, like RMI, with a garbage collected language, such as Java, you introduce the problem of remote references to locally dead objects. A server's garbage collector cannot simply collect remote objects when they are locally dead, because some client may still be using them. Not collecting these remote objects at all can lead to an object retention memory leak, because they may have no remote clients. Java's distributed garbage collector framework solves this problem by associating a lease with each remote object. Each time a client unmarshals a proxy, it requests a lease from the remote object's server. Before that lease expires, the client must renew it, or the server is free to collect the remote object.

6.2.3 Registry

A client requires a proxy to invoke operations on a remote object; in general, it acquires that proxy from an RMI registry. The RMI registry (`rmiregistry`), which listens on the well-known port 1099, binds a remote object's name to a proxy.

To bootstrap an RMI application, the client and server must share an interface that extends the `Remote` interface and defines a set of methods, and the client must statically know the URL of a host that is running `rmiregistry` and has exported the named remote object that implements the shared interface.

6.3 The MAGE Runtime System

MAGE modifies and extends the RMI runtime to handle mobile objects. In this section, we motivate and explain each of the required changes. Table 6.2 summarizes the MAGE services and the maps they maintain. The `InetSocketAddress` class wraps a IP address, port pair. Below, we discuss each service.

Service	Map
MAGE registry	id → `InetSocketAddress`
	`InetSocketAddress` → `MageRegistry`
class server	name → class-bytes
listener	tag → result
VM port	host → port
resource manager	`String` → `Object`

Table 6.2: MAGE Maps

6.3.1 Invocation Server

The MAGE invocation server's behavior is a superset of that of the RMI invocation server: it must handle the movement of remote objects. Upon receipt of an invocation, the MAGE invocation server must check whether the target mobile object is present, since another invocation may have moved the mobile object after the current invocation was dispatched. If the object has moved, the invocation server throws an exception, which restarts the find phase at the invoker. Otherwise, it checks whether it is the invocation's execution target. If not, it unexports the mobile object from the invocation server, adds the mobile object to the invocation, and forwards the invocation to the execution target. We describe the implementation of MAGE's invocation protocol in Section 6.5.4.

6.3.2 Distributed Garbage Collection

Since RMI remote objects cannot move, the RMI DGC client need only renew each lease with the DGC server that issued that lease. MAGE's mobile objects introduce the problem of notifying the DGC server of a mobile object's *current* host. When a DGC client attempts to renew the lease on a host that no longer hosts a mobile object, that host's DGC server informs the client that the object has moved. When so notified, MAGE DGC clients follow their mobile object's chain of forwarding pointers and update the DGC server to which they will direct their next renewal attempt to their object's current location.

6.3.3 Class Server

Sun's Java tutorial gives the code for a class server, which listens for and responds to requests for class definition, or class bytes [69]. Java's marshaling, aka serialization, libraries can annotate outbound objects with the URL of the codebase that contains class definitions that do not exist at the unmarshaling host. The unmarshaler uses this URL to contact the class server. In Java, such annotation is needed when a parameter passed to a remote method is an instance of a class unknown at the unmarshaler. This occurs when a client passes actuals that are instances of a local class that implements an interface or subclasses a class in the target remote method's signature, which the client and server necessarily share. MAGE leverages these annotations to propagate the class definitions of mobile objects, as they sojourn in the network.

6.3.4 MAGE Registry

In addition to relying on `rmiregistry`, the MAGE registry adds two components — `MageRegistryServer` and `MageRegistryImpl`. `MageRegistryImpl` maintains a map of `ObjID` identifiers to forwarding pointers. There is an entry in this map for each mobile object that has either been invoked from or visited the machine on which the singleton `MageRegistryImpl` is running (Chapter 5).

`MageRegistryServer` implements MAGE's find operation. When following a chain of forwarding pointers, it often needs to contact the same remote `MageRegistryImpl` instances over and over. To improve performance, `MageRegistryServer` caches proxies to remote MAGE registries in a map that binds an IP address and port pair to a proxy.

Upon startup, the MAGE registry reads an initial list of MAGE VMs into an instance of `HashSet<String>`. This set tracks the universe of MAGE VMs. The complement mobility attribute operator is defined in terms of this set, which it accesses via `MageRegistryServer.getVMs()`. A limitation of the current implementation is that this list is static (Section 6.6).

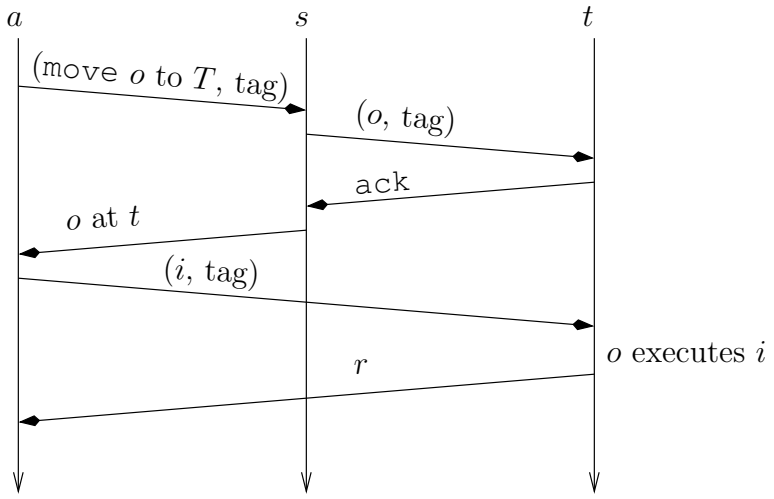


Figure 6.1: Protocol Without Listener

6.3.5 Invocation Return Listener

The point of MAGE is to control object mobility. Often the mobile object o that is the target of an invocation is found at s, not at the desired execution engine $t \in T$. The question is how to move it to t. MAGE could either first explicitly move o to t or make the move implicit to the handling of an invocation.

Figure 6.1 depicts a protocol that explicitly sends a move message. First, a sends an explicit move message for o to s. The move message contains the T that a's mobility attribute calculated so that s can use it as an input in the application of o's component mobility attribute, if one is bound. It also contains a globally unique tag. To prevent a from starving, the target t immobilizes o until it receives the invocation from a identified by the tag. A component mobility attribute may select any $t \in T$ or even override a's T altogether, so s must inform a which t was selected, with the message "o at t." Then a makes an RPC invocation on o at t, by sending the pair (i, tag) to t and awaiting the result r as shown. The target t uses the tag to identify the invocation that frees o to move again. While o is immobilized, it can execute operations that do not require it to move.

This solution requires a to block until it receives r and the addition of an explicit, distinct move message to the invocation protocol, and six network messages. To prevent starvation, it also requires that o be immobilized at t until a's invocation arrives. Preventing starvation

comes at a cost: what to do if the invocation never arrives? There are four responses: 1) allow starvation; 2) a could resend the invocation after a timeout if it does not receive a result[5]; 3) free o after a timeout period; or 4) accept that o may simply be immobilized indefinitely. MAGE's current implementation uses response 4, because immobility does not impact correctness and allows an application to progress, albeit at some performance cost. Empirically verifying this design decision is future work.

Figure 6.2 depicts the protocol that results when the move is implicit to an invocation. Here, we require a listener, as we do not wait to directly tell a which t was selected. First, a acquires a globally unique tag and an associated future from its local listener l. A future is an object that will hold the result of a future computation, here an invocation on o at t [39, 60]. Then a sends its invocation together with the tag to s the host at which a found o. The host s then marshals the mobile object o, builds the message (i, tag, o) and sends it to t where i executes. The host t then sends the result r together with the tag to l. The listener l uses the tag to put the result in the future identified by the tag.

A *future* is a mechanism that allows asynchronous RPC calls. When the caller invokes an asynchronous RPC call that supports futures, the caller immediately receives a future rather than blocking on the call. The caller is then free to perform other work in parallel to the remote call. At any time, the caller can check the future to see if the result has arrived or simply block on the future when the caller has exhausted the available concurrency [39, 60]. Thus, a future allows a to exploit the network latency of an RPC to perform other work in parallel instead of simply blocking. If a has nothing else to do and wishes to make a synchronous call, it simply immediately calls the future's blocking method for acquiring that result. The target execution engine forwards exceptions, if they occur through r. If, for some reason, o never executes and nothing is ever returned, a will block indefinitely on its future, just as the caller would do in a standard RPC call that never returned.

In addition to providing the parallelism of a future, this approach has other advantages. The actor a's interactions with l occur within an shared address space; they are memory

[5]This approach is closely related to the general problem of RPC semantics in the face of failure, *viz.* the choice among at-least-once, at-most-once, or exactly-once semantics.

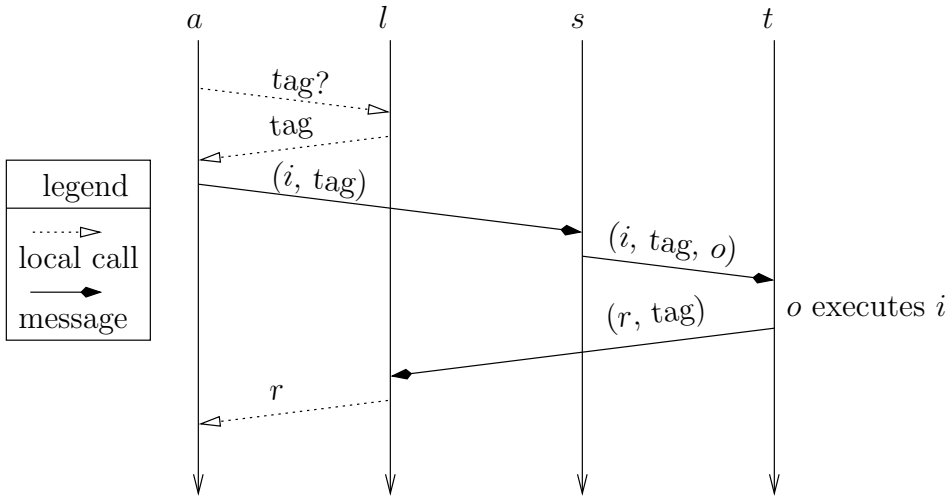


Figure 6.2: Listener Protocol

reads and writes. Thus, although this approach requires the listener l, it requires fewer network messages, three vs six; it does not require an explicit move message: nor must it immobilize o while it waits for a's invocation i, but instead can execute i immediately upon o's arrival. For these reasons, the current MAGE implementation uses a listener. The resulting invocation protocol is described below in Section 6.5.4.

6.3.6 VM Port Discovery

By default, RMI's invocation server listens at an ephemeral port assigned by the operating system, to ease running multiple invocation servers on a single host. This causes no problems for RMI since, by convention, an invoker bootstraps by downloading a proxy from the `rmiregistry` running on that host. The `rmiregistry` binds to the well-known port 1099 and each proxy contains the port of its remote object's invocation server.

When running multiple MAGE invocation servers on a single host, MAGE cannot simply extract the invocation port from a proxy for the mobile object o, as RMI does, because of the MAGE invocation protocol may require o's current host to forward o to another invocation server whose port is unknown to both o's invoker and current host. To partially solve this problem, MAGE provides a port discovery mechanism.

This discovery mechanism allows a client to specify a target location using only a host

Listing 6.1: Resource Manager Usage

```
HashMap loadByCPU = (HashMap) rm.get("CPU");
Integer load = (Integer) loadByCPU.get("sytem-of-interest");
```

address and no port. When a MAGE invocation server receives an invocation whose target is only a host address and it does not already know that host's default port, it learns the port at which that host's invocation server is listening as follows: It contacts that host's `rmiregistry` to get a proxy to that host's VM port server. That proxy supports `getPort()` which returns the address of the default invocation server on the machine on which the VM port server is running. This solution is partial because it designates one of the n MAGE instances sharing a machine as the default invocation server, and does not provide a general way to discover all of their ports.

6.3.7 Resource Manager

The MAGE resource manager is a subclass of Java's `Hashtable` that implements `Remote` and therefore allows remote queries and updates. The MAGE resource manager is loosely typed: for maximum flexibility, it maps `String` to `Object` instances. A programmer nests hash tables to store hierarchical data. Thus, a resource is anything that can represent itself as a graph of Java `Object` instances.

In Listing 6.1, looking up "CPU" returns a hash table that maps a host name to a string containing that host's load average. The application developer must have previously populated the queried `rm` with the requisite load data.

Each JVM running MAGE publishes a resource manager. MAGE does not automatically provide any resource information in its resource manager, instead leaving that task to the application developer. The purpose of the resource manager is to provide a convenient API for communicating resource state or handle to a resource to a policy that a developer embodies in a mobility attribute.

6.4 Primitives

In this section, we first present the challenges in implementing MAGE's mobile object and proxy primitives, then the solutions we implemented.

The challenges faced in implementing MAGE's mobile object follow:

Marshaling When an RMI remote object is an actual in an invocation, RMI marshals a proxy to the remote object, not the remote object itself. MAGE needs to support both this behavior, when a mobile object is an actual, as well as marshal the mobile object itself for movement.

Static Fields In standard Java, a class and all of its instances share a single address space. Thus, static fields, field defined on the class itself, not its instances, are a convenient mechanism for sharing data across instances. Mobility complicates the story: the instances of a class are no longer guaranteed to share an address space, and the class definition itself must exist within every address space that contains an instance.

Direct References To move the mobile object o to t, MAGE marshals a clone of o, sends that clone to t, unexports o, then updates the forwarding pointer for o to point to t, thereby implicitly updating all local proxies for o. Local references to the instance of o before it was cloned present a problem: any writes to this now dead clone are lost.

Initially, MAGE modified `rmic`, the RMI compiler, to produce MAGE proxies. In Java 5, RMI deprecated `rmic` and replaced it with the dynamic generation of proxies that support the signatures defined by an application's remote interface, but delegate the bulk of their implementation to an instance of `RemoteObjectInvocationHandler`. This class uses reflection to make the remote call. This abstraction simplified the implementation of MAGE proxies: the logic is now concentrated in `MageInvocationHandler`.

In RMI, the `Remote` interface defines only a type recognized by `rmic` and now by the dynamic proxy generator[6] to signal the creation of a proxy. `MageMobile` defines the bind methods that MAGE proxies must support. Ideally, `MageMobile` would obviate `Remote`

[6]The class `magesun.rmi.server.Util` in MAGE.

in a MAGE application, but that would require changing the dynamic proxy generator to recognize `MageMobile` instead of, or perhaps in addition to, `Remote`. This work, and its attendant testing, has not been done in the current implementation.

6.4.1 Mobile Objects

A language's support for mobility ranges from strong to weak. Under strong mobility, both a mobile component's code and execution state moves; under weak, only a mobile component's code moves [16]. On this continuum, Java is fairly weak, since it does not move stack or register state, although it does move heap state.

On the server side, an RMI remote object registers with RMI's invocation server and can receive and execute invocations from remote clients. In any remote procedure call implementation, client and server must share an application-defined interface through which they can communicate. Here, `AppIface` is that interface. RMI remote objects simply implement `AppIface`, which extends RMI's `Remote` interface. MAGE mobile objects, in contrast, must both implement `AppIface` and inherit from `MageMobileObject`, as shown in Figure 6.3. We discuss `MageMobile` in Section 6.4.2, which follows.

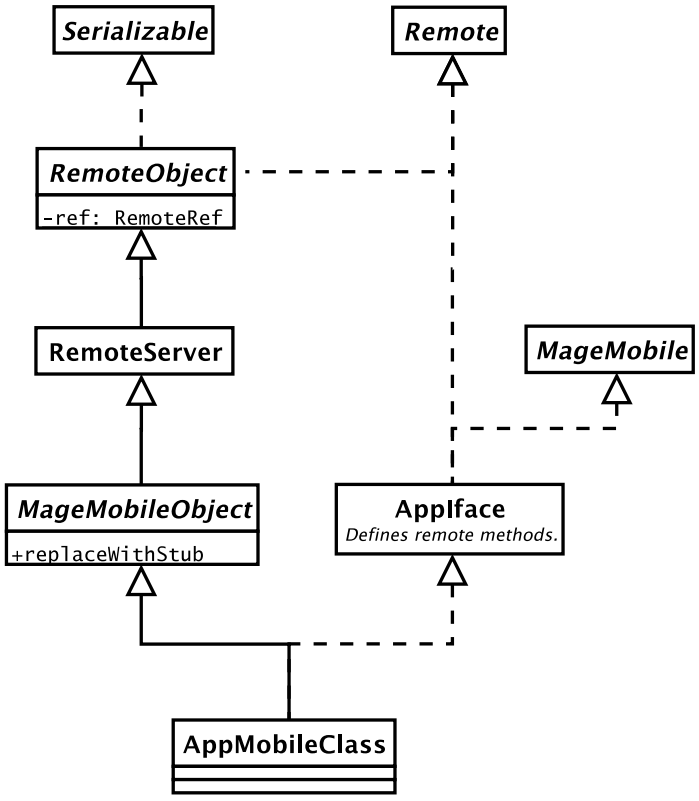


Figure 6.3: MAGE Mobile Object Class Diagram

`MageMobileObject` serves two purposes: 1) it defines the behavior that allows a programmer to lock the mobile object to a host as well as supporting the binding of

mobility attributes directly to a component (Section 3.6); and 2) it defines a type and a field that MAGE uses to control the marshaling of a MAGE mobile object. RMI uses Java's serialization library's `replaceObject()` method to override that library's default marshaling behavior. When a remote object is marshaled, its associated RMI proxy class (next section) is marshaled in its place.

For a mobile object to move, it must be marshaled, not its proxy, so MAGE must override this behavior. However, a MAGE mobile object potentially moves only when it is invoked, not merely referenced. Thus, when a mobile object is a merely a parameter in a remote call, MAGE must support RMI's default behavior and marshal that mobile object's proxy, not the object itself. MAGE uses `MageMobileObject` and its `move` field to distinguish these two cases. While a mobile object resides on a particular host, its `move` field is false. When MAGE marshals a mobile object for a move, it sets its `move` to true. This field remains true in the now-dead version of the mobile object left behind by the move. Threads in active mobile objects use this fact to stop and release the object for garbage collection.

To complete the discussion of Figure 6.3, we note that RMI's `RemoteObject` class implements the `java.lang.Object` behavior for remote objects and RMI's `RemoteServer` class is the common superclass to server implementations. `AppIface` denotes the application interface that defines the remote methods that instances of `AppMobileClass` can receive. We discuss it further in the next section.

In Java, a static field is a field shared by all instances of a class in an address space. Obviously, maintaining field sharing becomes costly when an object can move and instances can move among address spaces. By default, RMI does not marshal static fields, thereby punting this sharing problem to the programmer. If the programmer is sufficiently motivated, she can write and *exclusively* use getters and setters that broadcast updates to VMs that are hosting an instance of the class, write custom marshaling via Java's `readObject` and `writeObject` methods to track the set of VMs where instances exist, and roll her own replication mechanism. MAGE follows RMI's lead, and leaves the handling of static fields in mobile objects to the programmer. By default, MAGE objects exist in only one namespace at a time.

Like an RMI remote object, a MAGE mobile object is kept alive by proxies held by its clients, renewing their leases with the DGC server of the mobile object's current host. When a MAGE mobile object moves, it unexports itself, which unregisters it both from a VM's invocation and DGC servers on the VM it is leaving[7]. On all systems, other than a mobile object's origin host, access to that mobile object is mediated by a proxy acquired from the RMI registry, so this action removes all references to the discarded copy of the mobile object on that VM, leaving it eligible for collection by the local VM's garbage collector. A thread running on a mobile object's origin host could hold a reference to such a discarded clone, and violate MAGE's invariant that a mobile object exist in only one namespace at a time. MAGE does not track and therefore cannot replace local references with proxies. Ideally, the compiler would warning about a local reference to a mobile object. Currently, MAGE relies on the programmer to respect the convention that they should eschew direct, non-proxy references to mobile objects[8]. To allow a mobile object's clients time to catch up with a mobile object after it has decamped to a new host, MAGE pins mobile objects, thereby making them unreapable for one lease renewal period, which defaults to 10 minutes. This default can be changed by setting the property `mage.rmi.dgc.leaseValue`.

6.4.2 MAGE Proxies

The remote procedure call paradigm, of which RMI is an instance, relies on proxies, or stubs, to marshal calls at the client for writing to the network. A proxy generator creates these proxies automatically, either statically via the now deprecated `rmic` compiler prior to Java 5 or dynamically since Java 5. In both cases, remote proxies are generated for classes that implement the `Remote` interface. Thus, RMI's `Remote` interface defines remote methods; it designates interfaces for which a proxy must be generated. Java remote methods must throw `RemoteException`. RMI's proxy generator enforces this constraint.

Unlike an RMI proxy, a MAGE proxy, in addition to remote methods, defines local methods that bind mobility attributes to the proxy. Since having these methods throw

[7] To unexport itself, a mobile object must acquire its move lock.
[8] More empirical investigation is needed to determine whether local references are a problem in practice.

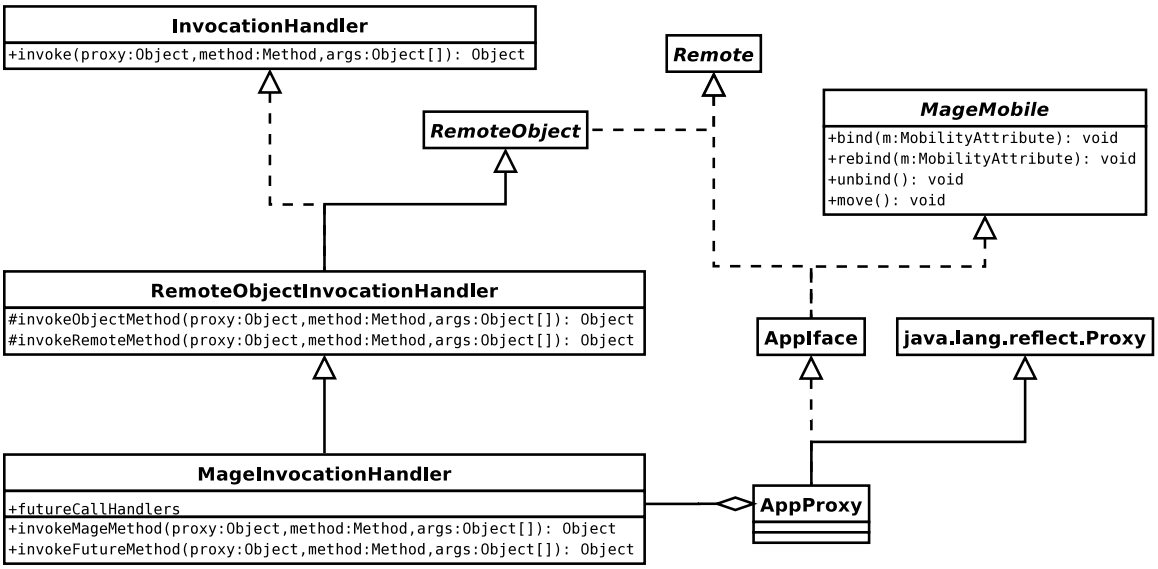


Figure 6.4: MAGE Proxy Class Diagram

`RemoteException` does not make sense, MAGE defines them in the interface `MageMobile` that does not extend `Remote`, as shown in Figure 6.4. `MageMobile` also defines a `move` method, whose purpose is to provide a hook on which to bind mobility attributes to handle system shutdowns. The idea is to make it easier for an application programmer to write a facility that, upon receipt of a system shutdown alert, calls `move` on all affected mobile objects.

In Figure 6.4, the shared application-defined interface `AppIface` allows client and server to communicate, as described in the last section. `AppProxy` is a Java 5 dynamically generated remote proxy that extends `java.lang.reflect.Proxy`, implements `AppIface`, and delegates invocation handling to `MageInvocationHandler`.

Figure 6.4 shows where methods are declared, not necessarily defined or overridden. The class `RemoteObjectInvocationHandler` defines the methods `invokeObjectMethod` and `invokeRemoteMethod`. The former method defines some of the local method calls declared by `Object`, like `toString()` and `equals()`; the latter method delegates the remote call to `UnicastRef.invoke()` which marshals the call, sends it to the remote server, and unmarshals the result. `MageInvocationHandler` overrides its inherited `invoke()` method to invoke `invokeFutureMethod` when the passed method object's

return type is `FutureMageCall` or, when `MageMobile` declared the passed method, to invoke `invokeMageMethod`, which implements `bind`, `rebind`, and `unbind`. The class `MageInvocationHandler` also overrides its inherited `invokeRemoteCall` to perform a MAGE call (Section 6.5.4) if a mobility attribute is bound to the proxy, or a standard RMI call otherwise.

`MageInvocationHandler` uses its thread pool of `FutureCallHandler` threads to implement futures (Section 6.3.5). When the return type of an incoming call is a descendant of the class `FutureMageCall`, an instance of `MageInvocationHandler` invokes its `invokeFutureMethod` method to handle the call. This method grabs a thread that makes the call on behalf of the calling thread, which is free to immediately return. When the remote call returns, the `FutureCallHandler` thread assigns the result, including exceptions, to the future. The calling thread can check this future at any time to see if the call has returned, or, when it has no further concurrent work to do, it can block on the future, as specified by `java.util.concurrent.Future<V>` [70].

The cost of supporting futures in MAGE is the cost of invoking the `isAssignableFrom` native method on `java.lang.Class`, and testing its result in a conditional.

6.5 Operations

In this section, we discuss the implementation of each of the classes of operations that MAGE introduces.

6.5.1 Find

Chapter 5 presents the design of MAGE's directory service, which uses forwarding pointers. MAGE caches a forwarding pointer to every mobile object that has ever been invoked from or has visited a host in its `MageRegistryImpl`. An invoker must download a proxy for a mobile object on which it wishes to invoke operations from that mobile object's origin server. When the proxy is unmarshaled, MAGE extracts the mobile object's identifier and adds an entry to the cache that points to the mobile object's origin server. When a mobile

object arrives at a host and is locally exported to receive remote calls, MAGE either adds or overwrites its entry with a self-loop, a forwarding pointer to the local host. When a mobile object departs, MAGE overwrites the mobile object's entry in `MageRegistryImpl` with a pointer to the mobile object's destination host.

The `MageRegistryServer` class, the local component of the MAGE registry (Section 6.3), implements a find method that straightforwardly walks the chain of forwarding pointers. The `MageRegistryServer` first looks up the target mobile object's last known location in the local `MageRegistryImpl` cache, then contacts that location's `MageRegistryImpl`. Each queried instance of `MageRegistryImpl` either replies with a forwarding pointer to the next `MageRegistryImpl` instance in the chain or signals that the target component is collocated with it by replying with its own location.

6.5.2 Bind

Mobility attributes bind to proxies to allow different invokers to apply different migration policies to a single mobile object. Mobility attributes also directly bind to mobile object to impose a shared policy on all invokers. Bound attributes decide where invocations on a mobile object should occur.

As noted in Section 6.4 on MAGE proxies, `MageInvocationHandler` implements client-side mobility attribute binding with a mobility attribute field and a accesser, mutator pair. Component mobility attributes are similarly implemented, but by the base class of all mobile classes in MAGE, `MageMobileObject`. As is convention in the Java JDK, `bind` throws `AlreadyBoundException` when a binding already exists. To overwrite a binding, the programmer must first call `unbind`, then `bind` or use `rebind`.

6.5.3 Mobility Attribute Operators

Mobility attributes define two sets, a set of valid locations at which to receive an invocation and a set of execution targets. Mobility attribute operators apply set operations, such as union, to these sets. Mobility attribute operators augment the set operations with `LEFT`

Listing 6.2: `starts`

```
1 public Set<String> starts(Method m) throws StartException {
2    return apply(opS,left.starts(m),right.starts(m));
3 }
```

and `RIGHT` binary operators that just return the named operand (Section 3.5 introduces these operators).

The `MobilityAttribute` base class implements mobility attribute operators. It contains fields to store operands and the S and T operators over those operands. As Listing 6.2 illustrates, `MobilityAttribute`'s default implementations of the `starts` method recurses into the operands. The `targets` method differs from the `starts` method in Listing 6.2 only in that it passes `opT` to `apply`. The `apply` method performs the specified operation. Essentially, `apply` is a switch statement over the set of operators. Complement is defined against a universe. Thus, each MAGE server must maintain a set of all MAGE servers, which `apply` uses to implement complement. Currently, this list statically defined in a configuration file read when each server starts.

6.5.4 Invocation

This section describes the implementation of an invocation on a mobile object in MAGE. MAGE augments the RMI invocation protocol in three ways:

1. It adds new messages, notably `MageCall` and the accompanying MAGE invocation;
2. It handles the movement of a mobile object in the presence of concurrent invocations; and
3. It supports the indirect return of results through a listener.

We begin with the format of the network messages. We then turn to invocation sequence diagram, which we use to frame the discussion of tasks assigned to the various classes that generate and handle the messages used. On the client, the bulk of the implementation resides in `MageInvocationHandler` and `UnicastRef`; on the server, in `TCPTransport` and

its parent `Transport`. We described `MageInvocationHandler` above (Section 6.4.2). In `UnicastRef`, the method `mageInvoke` loops over finding the mobile object, then attempts to make the invocation. If either the find or the invocation fails, because the object moves, `mageInvoke` restarts the loop, until a configurable starvation bound is exceeded. `TCPTransport` recognizes `MageCall` and hands off the stream to `serviceMageCall` in `Transport`. This method unmarshals the header of the incoming MAGE invocation, unmarshals and exports the mobile object if it is in the invocation stream, then applies the component mobility attribute if one is bound. Finally, `serviceMageCall` either dispatches the invocation or unexports and forwards the mobile object to its execution target. We close with a discussion of the algorithms to realize movement in the presence of concurrent invocations. The listener portion of the protocol was previously discussed in Section 6.3.5.

The most notable implementation challenges are:

1. Java objects cannot move without their classes, which may not be defined in the destination JVM;
2. Mobility in the presence of n simultaneous invocations; and
3. Component mobility attributes must be efficiently applied.

The limitations of the current implementation include

1. MAGE abuses exceptions to restart its invocation protocol and wastefully re-marshals the call at each restart;
2. Direct local calls on a mobile object are invisible to MAGE, so MAGE *can* move an object that contains running threads; and
3. Self-moves, moves triggered by a mobile object calling a mobility attribute mediated operation on itself, are restricted to void methods that lack in-out parameters.

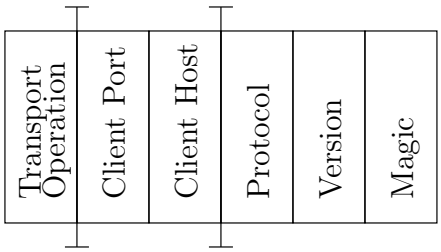
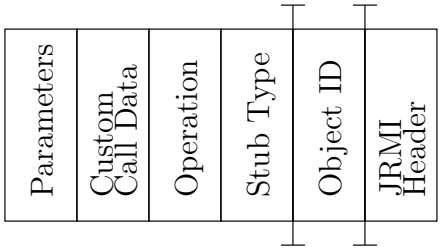
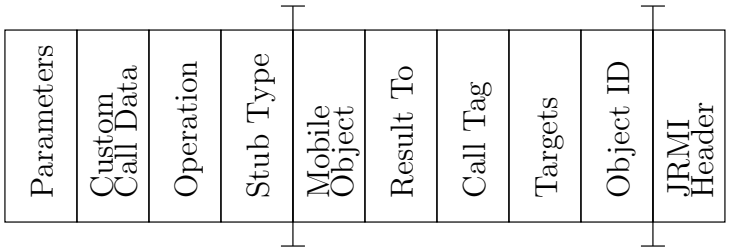


Figure 6.5: JRMI Header



(a) RMI Invocation



(b) MAGE Invocation

Figure 6.6: Invocation Formats

Message Formats

Java's RMI can run over various protocols, notably tunneled within HTTP and directly on TCP/IP. Figure 6.5 depicts the format of RMI's transport header[9]. The `Protocol` field is one of `SingleOp`, `Stream`, or `Multiplex`. The `Transport Operation` field is one of `Call`, `DGCAck`, or `Ping`. The latter two operations are used by Java's distributed garbage collection service. The bracketed fields — `Client Host` and `Client Port` — are not sent when the protocol is `SingleOp`. MAGE uses Java's JRMI header unchanged, but adds `MageCall` as an additional transport operation.

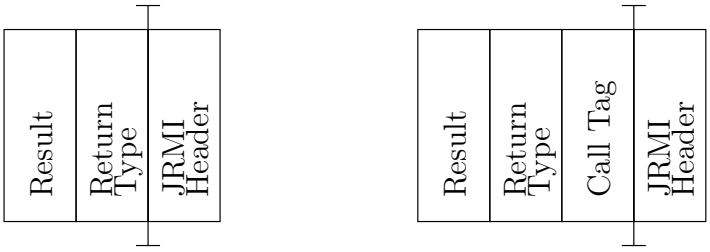
[9]Based on the content of RMI's "transport" header, it would have been better called a session header in terms of the OSI model.

Figure 6.6 presents the format of the invocation messages that RMI and MAGE use. MAGE adds four fields to the RMI invocation message — Targets, Call Tag, Result To, and Mobile Object. The targets field stores the client's target set created when the client applied its mobility attribute. Component mobility attributes (CMA), which bind mobility attributes directly to the component, uses the targets field as an input to the application of the component mobility attribute. The MAGE listener service requires the "Call Tag" and "Result To" fields. After the call executes on a server, that server uses the "Result To" field to determine where to route the result. The listener collocated with the invoker uses the call tag to route the result to the invoking thread. Finally, the mobile object field stores a marshaled representation of the target mobile object itself.

Figure 6.6a and Figure 6.6b are separated into three parts according to which class processes the relevant fields. From the right, in package `magesun.rmi.transport`, `tcp.TCPTransport` handles the JRMI header in both RMI and MAGE. `Transport` handles the Object ID field which identifies the remote, or mobile, object on the receiving server. When receiving a MAGE call, it also handles the additional MAGE fields. Finally, in package `magesun.rmi.server`, the methods `dispatch` and `mageDispatch` of class `UnicastServerRef` determine which stub type to use, then use the operation field as a key into their hash table of remote methods.

RMI's custom call data field allows an application to add custom call data to its invocations. To add custom call data, the programmer must override `UnicastRef`'s `marshalCustomCallData` method; to restore that data at the server, the programmer must concomitantly override `unmarshallCustomCallData` in `UnicastServerRef`. MAGE does not use this field because it is accessed too late in the processing of an invocation. In particular, the mobile object, if sent, must be instantiated and exported before it can receive calls. The custom call data field is not processed until after the call has been dispatched, at which point it is too late to export the target mobile object. It is also more efficient to handle the MAGE fields as early as possible when processing a call.

All marshaled objects, in particular those in the parameter and mobile object fields, are annotated with the URL of the invoker's `classserver`. If a class is not defined locally,



(a) RMI Return (b) MAGE Return

Figure 6.7: Invocation Return Formats

then `URLClassLoader` uses this annotation to contact the invoker's `classserver` to acquire the requisite class definitions, as described in Section 6.3.3.

Figure 6.7 depicts the formats of the RMI and MAGE invocation returns. A MAGE listener uses the call tag field to either notify a sleeping invoker or to place the result where the invoker can retrieve it.

Invocation Sequence Diagram

Figure 6.8 contains the sequence diagram for the invocation phase of the FI, the MAGE invocation protocol. From left to right in synopsis, the client a invokes its target mobile object o found at the server $s \in S$, the set of valid starting locations. The server s then unexports o, adds it to the invocation, and forwards both to the target server $t \in T$, the set of target execution hosts. The server t unmarshals the mobile object and exports it, before dispatching the call to it. The chain of local calls on t that eventually reaches `MageMobileObject`, the ancestor of all mobile objects in MAGE, represents this sequence of events.

From left to right in detail, the ApplicationClass contains the running code that invokes a method on the mobile object o by invoking a method of the desired name on Proxy. In JDK 1.5, the Proxy class is dynamically generated for classes that implement interfaces that are descendants of `Remote`. It delegates all calls to its `InvocationHandler`, `MageInvocationHandler`, instance through the invoke method. The invoke method of `MageInvocationHandler` recognizes four sorts of invocations:

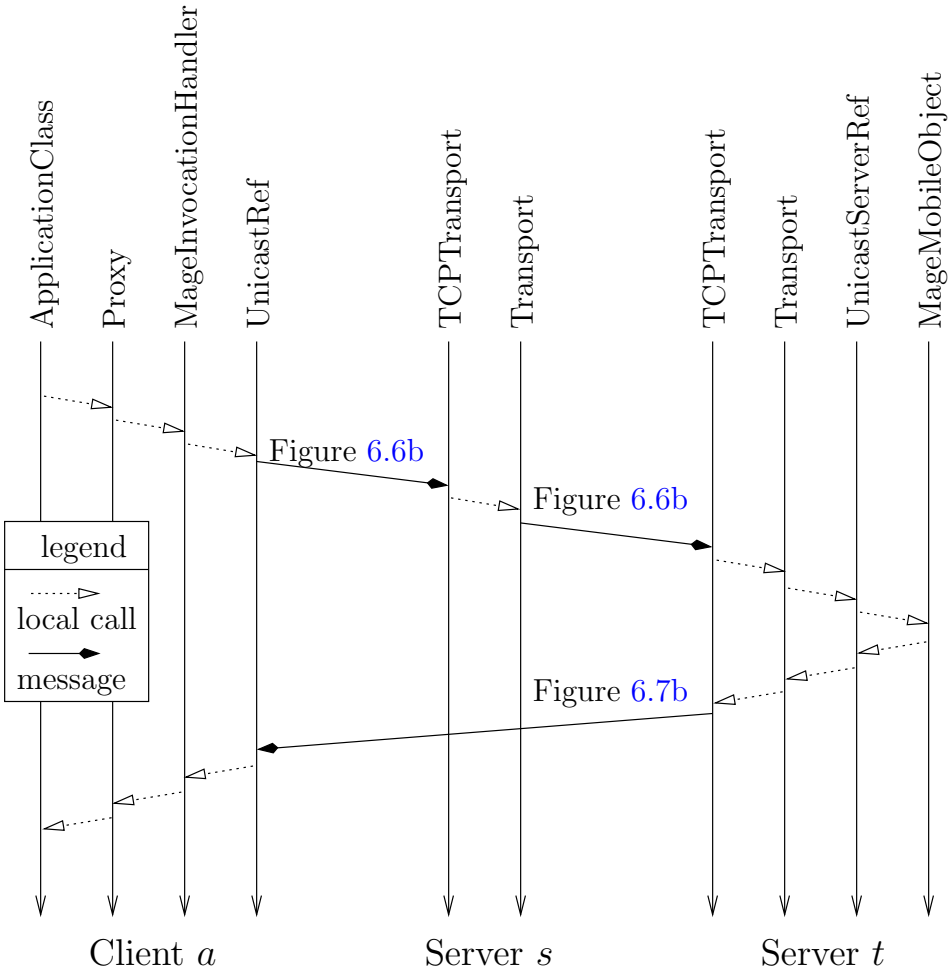


Figure 6.8: MAGE Invocation Sequence Diagram

1. Local method calls defined by `Object`, such as `equals`;

2. Local MAGE method calls, such as `bind`, which sets the `MageInvocationHandler`'s mobility attribute field;

3. Asynchronous MAGE remote calls, if the invocation's return type is a descendant of `FutureMageCall`; and

4. Synchronous MAGE remote calls, if a mobility attribute is bound.

The local MAGE methods are documented in Section 6.5.2. Future calls, documented in Section 6.4.2, simply use a thread pool to make synchronous MAGE remote calls.

Algorithm 6.1 `mageInvoke`

Input: $id, ma,$ mobileObject, method, parameters

```
1:  i := 1
2:  repeat
3:    try
4:      T := apply(s, method, ma) // Algorithm 6.2
5:      return executeCall(id, T, mobileObject, method, parameters) // Algorithm 6.3
6:    catch NoSuchObjectException, MovingException
7:      s := find(id)
8:      i := i + 1
9:  until i > starvation bound
10: throw CallStarvationException
```

Algorithm 6.2 `apply`

Input: $s, ma, method$

Require: $ma \neq$ `NULL`

```
1: S := ma.starts(method)
2: if s ∉ S then
3:   throw StartException
4: T := ma.targets(method)
5: if T = ∅ then
6:   throw TargetException
7: return T
```

`UnicastRef`'s `mageInvoke` method handles synchronous MAGE remote calls. This method wraps its call to `executeCall` in a loop and a try-catch block that restarts the call, subject to MAGE's starvation constraint, in the event that the target mobile object moves before the call succeeds.

Algorithm 6.1 depicts `mageInvoke`'s pseudocode. It optimistically assumes that the mobile object has not moved, and applies the mobility attribute ma by calling `apply` on line 4. In Algorithm 6.2, the `apply` method first checks that $s \in S$, then generates T. If the execution fails, Algorithm 6.1 follows the chain of forwarding pointers (the find phase of FI) to locate the target mobile object o on line 7. Each time `mageInvoke` must find the mobile object, it reapplies the mobile attribute.

Algorithm 6.3, implemented in `StreamRemoteCall`, actually sends the call and waits for a reply. It first acquires a call tag from its listener, then marshals the call, before blocking on listener's getReturn on line 5. When it receives an invocation return, the listener

Algorithm 6.3 `executeCall`

Input: id, T, method, parameters
Require: listener $\neq$ `NULL`
1: tag := listener.getTag()
2: resultTo := localhost
3: i := marshal(id, T, tag, resultTo, mobileObject, method, parameters)
4: send i to s
5: result,resultFrom := listener.getReturn(tag)
6: forwarding pointer := resultFrom
7: **return** result

unmarshals the result, then returns from getReturn to awaken the invoking thread. If the result is an exception, it percolates up to the invoker. Otherwise, on line 6 `executeCall` updates its forwarding pointer, which is realized as a `LiveRef` instance that contains the IP address and port of o's new location, thus collapsing the invoker's chain of forwarding pointers.

Algorithm 6.1 appears to be vulnerable to a time-of-check, time-of-use (TOCTOU) race: the target mobile object o could move before the invocation reaches it. However, this race is not actually a problem: if o moves, it will not be at the server s when the invocation arrives and s will either throw a `NoSuchObjectException` or `MovingException` to the invoker. When `mageInvoke` catches either of these exceptions, it restarts the invocation subject to the starvation bound.

At server s, an instance of `TCPTransport` listens for incoming invocations. When it receives a message, it processes the JRMI header, and identifies the message as a MAGE invocation, it calls its `serviceMageCall` in its superclass `Transport`. Algorithm 6.4 depicts `serviceMageCall`. On line 1, s extracts the named data from the invocation i. Then s checks whether i contains the mobile object o.

In the scenario of Figure 6.8, o is not in i, so s checks whether o is local on line 3. If o is not local, then either there was an error, or o moved before the i reached s and the invoker must re-find o. On line 7, s throws `NoSuchObjectException` to report this case to the invoker. In scenario of Figure 6.8, o is on s, so `serviceMageCall` looks o up on line 4 and records that it did so by setting newlyArrived to false on line 5. Then `serviceMageCall`

Algorithm 6.4 `serviceMageCall`

Require: $\neg(o \in i \wedge \text{isLocal}(o))$
Input: i // An incoming invocation
1: $id, T, o, \text{resultTo}, \text{tag} = i.\text{unmarshal}()$
2: **if** $o =$ **null then**
3: **if** isLocal(id) **then**
4: $o :=$ lookup(id)
5: newlyArrived := false
6: **else**
7: throw `NoSuchObjectException`
8: **else**
9: export id, o with mlock = 0 and call count = 1 // after export, isLocal(id) = **true**
10: newlyArrived := true
11: $T :=$ `applyCMA`(o, T) // Algorithm 6.5
12: **if** self $\notin T$ **then** // Forward the invocation.
13: **atomic**
14: **if** o.immobilizations > 0 **then**
15: throw `ImmobilizedException`
16: **if** o.mlock $= 1$ **then**
17: throw `MovingException`
18: o.mlock := 1
19: **if** newlyArrived **then**
20: o's call count := o's call count - 1
21: block until o's call count = 0
22: unexport id, o
23: send id, **null**, o, resultTo, tag, i to $t \in T$
24: **return**
25: **else**
26: **atomic**
27: **if** o.mlock $= 1$ **then**
28: throw `MovingException`
29: **if** ¬newlyArrived **then**
30: increment o's call count
31: method, parameters $= i$.unmarshal()
32: result = dispatch(o,method, parameters)
33: **atomic**
34: decrement o's call count
35: send tag, result to resultTo

calls `applyCMA` to apply o's component mobility attribute, if one is bound, on line 11. Algorithm 6.5 captures the behavior of `applyCMA` in pseudocode.

The call count of o tracks the number of threads executing in o. On line 12, s checks whether it is itself an acceptable execution target. If it were, s would atomically check whether there was a waiting mover and increment o's call count on lines 26–30, before executing the invocation, lines 31–35.

We are considering the case in which s forwards o, which it accomplishes on lines 13–24.

Algorithm 6.5 `applyCMA`

Require: $o \neq \textbf{null} \wedge o.m \neq \textbf{null} \implies o.Op \neq \textbf{null}$
Input: $o, T \subseteq H$
1: **if** $o.m \neq \textbf{null}$ **then**
2: $T_c := o.m.\text{targets}()$
3: $T := T\ o.Op\ T_c$
4: **if** $T = \emptyset$ **then**
5: throw `TargetException`
6: **return** T

The server s first checks whether o has been immobilized. Then it attempts to acquire o's move lock to prevent any new threads from entering o by causing attempts to increment o's call count to throw `MovingException` on line 28. After setting o.mlock, s waits for the threads already in o to drain at line 21. There can only be one mover at time, so after a thread has acquired a mobile object's `mlock`, subsequent attempts to acquire it throw `MovingException` on line 17. Note that the move lock is needed only for side-effects confined to the mobile object itself, since other side-effects written by a thread running in the context of a dead clone will not be lost.

Server t's `TCPTransport` instance also uses `serviceMageCall` to handle the incoming invocation. This time the mobile object is in the invocation stream, so t exports o on line 9. This export atomically sets o's mlock to 0 and increments its call count to prevent the mover from starving as could occur if another thread moved o before the client a could execute its method on lines 31–35.

The method `mageDispatch` in `UnicastServerRef` implements dispatch on line 32. The method `mageDispatch` calls the method named by the invocation o via Java's reflection facility. It differs from it RMI progenitor in that it must handle both direct and indirect, listener mediated returns. In Figure 6.8, the invocation is eventually dispatched on the application's mobile object, which `MageMobileObject`, the ancestor of all mobile object classes, represents.

When more than one invocation for the same object arrives simultaneously, one of the invocations will win the race to lock the object on line 13. Whichever does will unexport the mobile object and move it. Each loser will report a `NoSuchObjectException` back

to its invoker.

A MAGE mobile object cannot move while it is has in-flight invocations that threads executing within it sent. MAGE derives this limitation from Java's weak mobility support: Java does not support thread migration. If it did, the MAGE listener could forward the results of in-flight invocations. So long as all calls made on a mobile object are proxy-mediated remote calls, the object's call count will be nonzero and will prevent that object from moving until all in-flight invocations return and the call count goes to zero, as described in Algorithm 6.4.

Direct, local, non-proxy-mediated calls on a mobile object are a problem, as described in Section 6.4.1. The problem occurs because a thread can hold a direct reference to a mobile object and thereby prevent the copy left behind after the object moves from being reaped. These calls do not increment the object's call count. Just as with the garbage collection problem, MAGE relies on convention to mitigate this problem: programmers must avoid local calls on mobile object and restrict themselves to proxy-mediated calls.

A MAGE mobile object moves itself by calling one of its own methods via a proxy whose mobility attribute picks a target set that does not include the current host. The invoking thread makes its call as usual and blocks in the listener. The host to which the object moves executes the call as usual, and forwards the result to the old host's listener. The problem is writes to fields in the dead clone. Any such writes must be forwarded to the mobile object's new host. When the method is void, side-effect free, and lacks in-out parameters, there is no problem: the invoking thread does not write to fields in the dead clone. The invoking thread can, of course, write to any other location in the application, such as a different mobile object.

When a mobile object moves, MAGE sets its `moved` field to true to marshal the mobile object itself and not a proxy to the mobile object, as occurs when the mobile object appears in a parameter list. The invoking thread could check this field, realize that the mobile object has moved and call a "fix-up" method on its proxy defined by `MageMobileObject` to forward the result to the mobile object. Of course, the result may not change the mobile object's state, in which case it can be dropped, or it may simply return to the remote invoker

that invoked the thread that caused the mobile object to move, in which case nothing special need be done. At present, MAGE relies on the programmer to restrict herself to self-moves triggered only by calls to void methods that lack in-out parameters.

6.6 Limitations

The current implementation of MAGE, like all implementations, is not ideal. Since MAGE was implemented, Java added its JMX API for management services [68] that obviates the MAGE resource manager. MAGE would better integrate with the Java ecosystem if it replaced its resource manager with one based on JMX.

MAGE's class server runs on its own port, rather than sharing a port with a single listener that multiplexes its messages with the other services. It should be merged with the others. Currently, many of MAGE's services run as RMI remote objects. MAGE's efficiency and scalability would be improved if they were directly integrated into MAGE framework.

MAGE relies on convention to prevent programmers from holding direct, non-proxy-mediated references to a mobile object and making direct, local calls on it. One possible way to enforce this convention is to require programmers to use a factory to instantiate mobile objects. This mobile object factory would return proxies on the origin host. The standard tactic to restrict instantiation to a factory is to make the constructor of the produced class private. This is not feasible for MAGE, since applications need to subclass `MageMobileObject`. Instead, the MAGE mobile object factory could track those objects that it creates and MAGE could refuse to move objects that it did not create.

Each MAGE server learns about other MAGE servers statically, via a configuration file. A dynamic broadcast mechanism for announcing new and departing MAGE servers should be incorporated, including a heartbeat to detect servers that are silently offline.

Currently the MAGE registry has two layers — the RMI registry and a RMI Remote object that implements the MAGE registry extensions and must be downloaded from the RMI registry. MAGE's registry functionality should be merged into a single service.

When a mobile object moves before an invocation reaches it, Algorithm 6.1 catches the

resulting exception, restarts the invocation protocol, and wastefully re-marshals the call. Further, its implementation abuses Java's exception handling mechanism to signal re-starting the invocation rather than an error [88]. A better solution would be to handle the problem at a lower level via a return type, not an exception, and avoid the redundant work.

6.7 Summary

In this chapter, we have described the implementation of MAGE. We compared the MAGE runtime to the RMI runtime, which MAGE extends. We described the implementation of MAGE's three primitives, followed by the operations that MAGE allows over those primitives.

Chapter 7

Evaluation

True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information.

Winston Churchill, 1874–1965

For mobility attributes to be at all practical, they must not impose too much overhead upon their user. In this chapter, we present two sets of benchmarks that compare MAGE against RMI:

1. micro-benchmarks that capture the invocation overhead of classic distributed programming paradigms realized as mobility attributes; and

2. a benchmark that quantifies the MAGE's marshaling overhead.

In Section 7.2, we present a novel variant of dining philosophers in which philosophers must eat and exploit mobility to satisfy their hunger. We describe a MAGE-based solution to this problem, which models philosophers as active mobile objects contending over chopsticks, also modeled as mobile objects. Activity and chopstick contention make this application a particularly demanding test of the robustness of the MAGE framework. We then describe and evaluate various different migration policies that philosophers can use to acquire food.

Connected via standard 100Mb Ethernet, our experimental testbed is heterogeneous, consisting of three machines: 1. a dual-processor Intel 500 MHz Pentium III with 512MB of

RAM, 2. a dual-processor Intel 2.80 GHz Pentium 4 with 1GB of RAM, and 3. an Intel Xeon 2.80 GHz with 2GB of RAM. Each machine runs Linux 2.6.20. We use Sun's JDK 1.5.0_07.

7.1 Baseline Measurements

The contributions of MAGE include a rich set of expressions for defining arbitrary mobile object and invocation placement policies, isolating those policies, and composing those policies (Section 1.3). The expressivity and separation of concerns MAGE offers is not free. These measurements quantify the overhead of the MAGE programming model. We present two sets of benchmarks here — micro-benchmarks of distributed invocations of passive objects and a marshaling benchmark.

We do not compare MAGE to mobile programming languages, such as Telescript [108] or D'Agent [36], here because, unlike MAGE, they make no attempt to isolate layout decisions. Their relationship to MAGE is analogous to that of assembly language to a 3rd generation language: They are lower-level and thus impose less execution overhead at the cost of a higher cognitive load on the programmer when writing and maintaining code.

No standard benchmarks or test suite exists for the programming models most closely related to MAGE (Chapter 8). Indeed, the related work all concentrates on illustrating their programming model using code snippets drawn from toy applications. Only SATIN [123] reports any quantitative measurements at all — the SLOC and bytes of the SATIN implementation and various applications, the times of micro-operations of launching a "Hello, world!" application on two machines, and the time to send and deploy an Ogg Vorbis codec. The SATIN authors offer no baseline against which to compare these numbers.

We begin with invocation micro-benchmarks because other costs will be either amortized over the life of an application (*i.e.* class-loading) or are specific to a particular application or environment, such as the cost of a policy that collocates an object with a resource like a printer or a database. Thus, rather than try to measure the cost of all the functionality MAGE can provide, we measure the overhead it imposes when it realizes RMC and the

classic invocation paradigms — RPC, COD, and REV[1]. Our goal is to demonstrate that MAGE's qualitative benefits do not come at too high a quantitative price. We include Java RMI in the invocation benchmarks as a baseline for comparison because (1) Java RMI is a well-known, standard mechanism; (2) the difference between Java RMI and MAGE RPC illustrates MAGE's overhead in the absence of mobility; and (3) other researchers can multiply these MAGE micro-benchmark results against the ratio of the performance of Java RMI in their environment to the Java RMI performance reported here to estimate how MAGE would perform in their environment.

The execution of a function call naturally divides into the overhead of invocation mechanism and the work done by the call. The work a call does is application specific and can take arbitrary time. Here, we seek to measure the overhead of the MAGE invocation mechanism. Thus, the invocation micro-benchmarks do nugatory work: they increment an integer. As a result, they shed no light on the question of whether the overhead of the MAGE invocation mechanism is independent of and constant in the presence of work. Marshalling return values is an easily duplicated and calibrated proxy for work. Thus, Section 7.1.2 uses marshaling of increasingly large return values to show that, while MAGE's overhead is not constant, it is a declining fraction of the total cost of an invocation as the work per invocation increases.

7.1.1 Invocation Micro-Benchmarks

MAGE adds three principal sources of overhead to a Java RMI invocation: the time needed to

1. evaluate the mobility attributes bound to an invoker's proxy and directly to the mobility attribute itself;
2. marshal the fields MAGE has added to the invocation message; and
3. manage mobility — specifically, a) move a mobile object, b) handle the possibility that

[1]These models were introduced and described in Chapter 3.

Listing 7.1: Invocation Measurement Test Class

```
1  package paradigm;
2
3  import player.Ball;
4  import mage.rmi.RemoteException;
5
6  public class BallImpl extends MageMobileObject implements Ball
7  {
8     private static final long serialVersionUID = 1L;
9     protected int cnt;
10
11    public BallImpl() throws RemoteException {
12       cnt = 0;
13    }
14
15    public void incr() throws RemoteException {
16       ++cnt;
17       return;
18    }
19 }
```

the target mobile object has moved since the invocation was sent, c) collect garbage, and d) return routing via the MAGE listener.

Item 1 — the time to evaluate a mobility attribute — can take arbitrary time, especially when that mobility attribute interacts with its environment via the resource manager. Thus, we employ the attributes tailored for each invocation paradigm. To minimize the cost of mobility, we statically deploy the test class. Since we do not need to bind mobility attributes directly to mobile object to model the above invocation paradigms, we do not bind mobility attributes directly to mobile objects. To avoid the overhead associated with items 3b and 3c, we restrict ourselves to single-threaded benchmarks, and we do not run the experiments long enough to evaluate the impact of garbage collection under MAGE.

Listing 7.1 contains the mobile test class, `BallImpl`. Because our focus is the cost the MAGE invocation infrastructure, this class does almost no work: it has a single integer attribute, which it increments.

Invocation Paradigm	Cold Invocation Time (ms)	Warm Invocation Time(ms)
JAVA RMI	2.03	1.28
MAGE RPC	12.69	3.33
COD	123.32	10.30
REV	124.27	12.37
RMC	150.45	13.35

Table 7.1: Invocation Measurements

For COD, an instance `BallImpl` migrates to the invoker's host. Once a test object arrives, the `incr` method is invoked and the results are returned to the invoker via the listener. For REV, we do the reverse. A local instance of `BallImpl` migrates to the remote host by instantiating a new clone at the remote host and discarding the old clone at the local host. The result is sent back to the local host. RMC is similar to REV except that the test object starts out and remains remote.

The measurements are contained in Table 7.1. We give cold and warm invocation times in the second and third columns, respectively. The reported numbers for both are an average of 10 runs. For cold, each run sets up a fresh server, downloads a fresh proxy, then times a single invocation. Thus, the cold invocation times show the one-time startup cost of priming the MAGE engine — loading the CPU and disk caches with relevant code and data as well as the cost of starting a connection thread that unmarshals and dispatches the invocation on the server. For the mobile paradigms, the startup cost to move the definition of the `BallImpl` to the target execution JVM dominates the total mean time.

The warm benchmarks set up the server, download a proxy, and make an initial call whose time-to-completion is ignored, before timing 10 consecutive calls. Dropping the first call approximates amortizing its cost without requiring a large number of runs. Thus, the warm times give a more accurate representation that MAGE applications will experience. Further, it is much easier to gather statistically significant data from long running servers. For these reasons, we now only discuss the amortized times.

We can see from Table 7.1 that the time reported for MAGE's implementation of the well-known distributed models. COD, REV, and RMC all move `BallImpl` prior to

executing `incr`. Thus, it is not surprising that their mean total time exceeds that of MAGE RPC, the immobile case, by an order of magnitude. COD is the fastest because its return, although listener-mediated, uses TCP/IP loopback and is entirely local[2]. REV is faster than RMC because the arriving object is registered with the target invocation server, `incr` is immediately executed, and the return is directly back on the socket used to deliver the invocation, unmediated by the MAGE invocation listener. RMC comes in last because it is both remote and listener-mediated.

When comparing a Java RMI call against a MAGE RPC call, we see that MAGE imposes overhead of 160% on a vanilla Java RMI call. To explain this delta, I ran the Java RMI and MAGE RPC invocation benchmarks 100,000 instead of 10 times each and ran the YourKit Java Profiler [121] against the results. The cost of marshaling accounts for most of this overhead. MAGE adds four fields to the Java RMI invocation header, of which three are non-primitive — the set of targets as an instance of `Set<String>`, the host to which to route the result as a `String`, and the mobile object itself. At the client, a MAGE invoker spends a factor of 3 more time than an RMI invoker to marshal an invocation. The MAGE server, however, spends still more time and is the bottleneck: the MAGE server spends a factor of 4 more time waiting to unmarshal and unmarshaling the invocation than the RMI server, slightly more than $\frac{1}{2}$ its total time. MAGE RPC's mean time-to-completion is 260% that of Java RMI. Thus, $\frac{3}{4}$ of $\frac{1}{2}$ or $\frac{3}{8}(260\% = \frac{13}{5}) = \frac{39}{40} = 97.5\%$ is unmarshaling overhead at the server. As a fraction of total overhead, $\frac{97.5\%}{160\%} = 61\%$ accounts for the majority of MAGE's overhead. Evidently, MAGE's implementation could benefit from the same sort of optimizations that have been proposed for RMI, in particular marshaling libraries that cache marshaled objects [74, 55].

Of the remaining and $160\% - 97.5\% = 62.5\%$ of MAGE's overhead, profiling reveals that MAGE pays, over and above Java RMI, for the cost of 1) string handling in the application of mobility attributes, 2) ensuring that sockets acquired from the socket pool are still alive by pinging the server-side, and 3) working around Java's partitioning of the

[2]MAGE could decrease this cost still further by optimizing for collocation as proposed at the close of Section 6.1.

table in which it stores server objects that remote clients can invoke (`ObjectTable`). For item 2, MAGE clients spend $\frac{1}{40}$ of the mean total time waiting for replies to their ping vs 0, *i.e.* unmeasurable noise, under Java RMI. Prior to Java 5, the `ObjectTable` was global within a JVM. With Java 5, Sun partitioned remote objects by invocation listener port, so that the clients of one listener could not invoke operations on remote objects exported at another listener within the same JVM. A mobile object may create a new invocation listener when it arrives at a server whose port is unknown to the proxies of existing clients. To work around this issue, MAGE searches the object table globally, at measurable cost.

Like almost all code, MAGE would obviously benefit from optimization. For instance, the set of hosts is likely to change slowly and predictably enough to be represented with bit strings. The current implementation focuses on being robust and maintainable. It has been written with a eye to Knuth's dictum — "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." [51] — perhaps with too generous an estimation of what constitutes a "small efficiency."

7.1.2 Overhead in the Presence of Work

Is the invocation overhead MAGE imposes independent of the work done by an invocation? The next benchmark seeks to shed light on this question. Marshaling is a simple example of generic work. Moreover, MAGE is built on top of Java RMI, so it uses the same marshaling library as RMI — `java.io.Serializable`.

Figure 7.1 shows how MAGE's overhead declines as the marshaling cost of a call increases. In this experiment, we return an increasingly large `ArrayList` whose elements are instances of a class that consists of four strings whose length is normally distributed about 40 characters, two `java.util.Date` instances, a **`long`**, an **`int`**, a **`double`** and a **`float`**[3].

If the MAGE overhead were independent of the work an invocation does, it would take a fixed amount of time and thus be a fast decreasing fraction of the time-to-completion of a MAGE invocation as the size of the returned `ArrayList` increases. Table 7.2 records

[3]This benchmark was inspired by a similar benchmark posted at http://daniel.gredler.net/2008/01/07/java-remoting-protocol-benchmarks/.

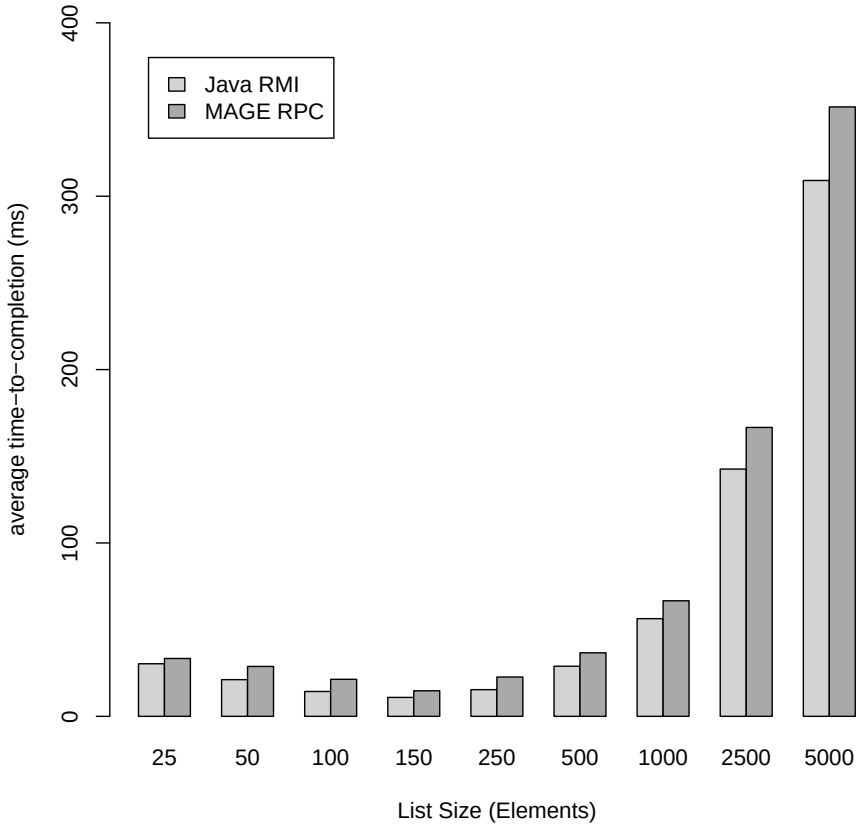


Figure 7.1: MAGE Marshalling Measurements

Elements	$\frac{\mu(\text{mage})-\mu(\text{rmi})}{\mu(\text{mage})}$
25	0.09098111
50	0.26457409
100	0.32884892
150	0.25898509
250	0.32108809
500	0.21121683
1000	0.15482159
2500	0.14406123
5000	0.12072428

Table 7.2: Mean MAGE Overhead Relative to RMI as Fraction of Total Time.

the mean overhead of MAGE relative to RMI as a fraction of MAGE's mean total time. As Table 7.2 makes clear, the MAGE overhead is not fixed, so it is not independent of the marshaling work. It is, however, a decreasing fraction of the total time.

7.2 Peripatetic Dining Philosophers

To exercise the MAGE framework and to illustrate the flexibility and concision of the migration policies it can express, we define "peripatetic dining philosophers," a novel variant of the dining philosopher's problem that requires mobility. We describe the implementation of our solution, which requires active mobile objects. We then present and compare a variety of migration policies, against both a fixed and changing backdrop of resource production.

Definition 7.2.1 (Dining Philosophers [25, 17])**.** Around a circular table, n philosophers rest, eat, and think. Each philosopher needs 2 chopsticks to eat, but there are only n chopsticks, one between each pair of philosophers. The *dining philosophers* problem is to devise an algorithm that the philosophers can follow that is

1. deterministic;
2. concurrent;
3. deadlock and livelock free;
4. fair;
5. bounded; and
6. economical.

Remark. "Bounded" means that both the number and size of messages in transit is finite; economical means that a philosopher sends a finite number of messages in each state — resting, eating or thinking.

We can capture the arrangement of philosophers and chopsticks in a ring. Figure 7.2 depicts the minimal ring.

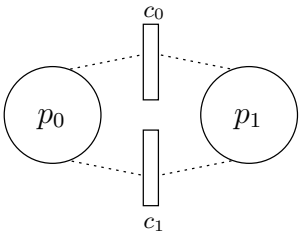


Figure 7.2: Minimal Dining Philosopher Ring

The dining philosophers problem models resource allocation in the presence of cycles in the resource graph: its philosophers are agents that require forks, a set of shared resources, whose cardinality is classically but not necessarily two. It does not model resource consumption.

What if philosophers actually needed to consume food in order to think? What if one fork were a streaming data source, like traffic crossing a backbone router, the other fork were an encrypted channel to the NSA, and the philosopher's food was the CPU required to forward a filtered version of the stream? In terms of the metaphor, we can replace the table around which the philosophers are seated with a set of cafés. In this new problem, philosophers may wish to move from one café to another, depending on food availability.

Definition 7.2.2 (Peripatetic Dining Philosophers)**.** The *peripatetic dining philosophers* problem is to satisfy the constraints of the dining philosopher problem, then maximize work per unit time, subject to the following rules:

1. Cafés produce food;
2. Philosophers must eat x servings before they can think, for $x \in \mathbb{N}$;
3. Both philosophers and chopsticks can move from one café to another; and
4. To eat, a philosopher and his chopsticks must share a café.

Remark. In contrast with the Evolving Philosopher Problem [53], the ring in which philosophers and chopsticks alternate does *not* change under peripatetic dining philosophers.

Maximizing work per unit time entails optimizing the layout of philosophers onto cafés, *viz.* finding an optimal migration strategy.

A philosopher can only work (think), *i.e.* perform filtering for the NSA as described above, when it has the requisite resources, *viz.* forks. Thus, we use the count of the number of times a philosopher enters its think phase as a proxy for the work it does.

Time-to-completion t, work per unit time w, and total work are related as follows:

$$wt = W \tag{7.1}$$

Thus, given a time budget t, maximizing w maximizes total work W; given work to do W, maximizing w minimizes time-to-completion t.

By extending dining philosophers to model resource consumption, peripatetic dining philosophers marries dining philosophers to the producer-consumer problem in the context of code mobility. Philosophers can now differ in their appetites; that is, how much food they consume when they are hungry. Cafés can produce food at different rates. When the demand for food at a particular café is too great, a philosopher has a reason to move to another café.

Peripatetic dining philosophers defines a family of problems. Two vectors parameterize this family for each café; one contains food production rates, the other starting amounts of food. Four vectors characterize each philosopher: one for consumption rates, starting café, time required for thinking, and the number of servings needed to sate the philosopher. The cost to move a chopstick and the cost to move a philosopher define two more dimensions. Finally, another group of problems is formed by producing each of these vectors with a possibly stochastic function.

In this chapter, we restrict ourselves to problem instances in which the vector of starting amount of food at each café is all zero and the philosopher consumption rate, time-to-think, and servings-to-sate vectors are all one.

Under peripatetic dining philosophers, the philosopher ring is mapped onto the set of cafés. Those mappings differ in terms of the number of neighboring philosophers in the ring that share a café. When two philosophers who share a chopstick in the ring share a café,

their chopstick can remain at that café. When neighboring philosophers are at different café's, they must send their shared chopstick back and forth between the two cafés. Clearly, chopstick acquisition is more expensive when the philosophers that share a chopstick are not collocated. Thus, any philosopher migration policy should seek to collocate philosophers that share a chopstick.

Dining philosophers is a generic and abstract problem that elucidates problems that arise when attempting to realize mutual exclusion. Systems composed of mobile objects also need mutual exclusion, so peripatetic dining philosophers is well-motivated. However, peripatetic dining philosophers does appear to strain the metaphor — why would anyone send chopsticks back and forth between two cafés? — until one asks why would anyone share chopsticks in the dining philosopher problem? In future work, we intend to consider modifying the dining philosopher ring by chopstick exchange to minimize chopstick movement as a way to minimize time-to-completion.

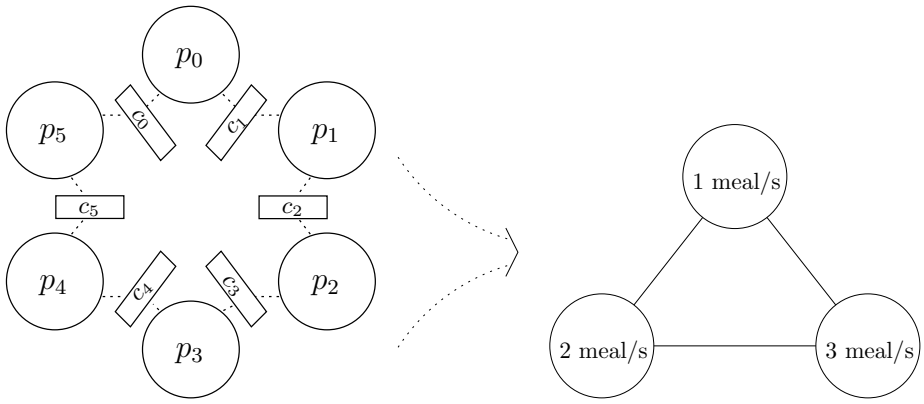
Example

Imagine that the two philosophers in Figure 7.2 have access to two cafés, A and B. Both café's start with 0 servings. A produces 1 serving/s, while B produces 2 servings/s. The two philosophers must each think, and therefore eat, 10 times. Both eat 2 serving/s.

If the philosophers start and remain at A, they will complete their work in 20s and 40 servings will go to waste at B. If the philosophers start and remain at B, they will complete their work in 10s and 10 servings will go to waste at A. Since the philosophers do not move, neither do their chopsticks.

In practice, the cost of chopstick and philosopher movement is greater than zero. Assuming movement were free, however, the philosophers minimize their collective time-to-completion when they dine at separate cafés. If one philosopher starts and remains at A and the other at B, then the philosopher at B will finish in 5s, while the philosopher at A finishes at 10s. Café B continued to produce food while the philosopher at A waited for food, so 10 servings are wasted.

Collectively, the philosophers require 20 servings. The two café's produce 21 servings in



(a) Six Philosopher Ring (b) Cafés

Figure 7.3: The Layout Problem

7s, which bounds time-to-completion from below. After the philosopher at B completes its work in 5 seconds, the philosopher at A still need to eat 5 servings. If it remains at A, sating its hunger will take an addition 5s, as in the scenario above. If it moves to B, it can sate its hunger in 3s. Under this migration strategy, the time-to-completion is 8s and 4 servings are wasted.

Of course, when the production rate of the cafés is fixed, as in this example, mobility reduces to optimal deployment, with some latency for discovering the café production rates. Mobility and migration policies become essential when café production rates change over time. Next, we use MAGE to empirically investigate the peripatetic dining philosophers problem, using different migration policies.

7.2.1 A MAGE Implementation

The peripatetic dining philosopher problem boils down the layout problem for which MAGE was designed: an optimal layout of philosophers maximizes work per unit time.

The MAGE implementation runs six philosophers. Figure 7.3a depicts the standard dining philosopher ring for six philosophers. The philosophers all request the chopstick to their right, then left, except p_5 who breaks the symmetry by requesting the chopstick to his left, c_0, before c_5. Each philosopher must eat 1 meal each time they are hungry. Figure 7.3b depicts the cafés, each labeled with their rate of food production. Note that each second,

Listing 7.2: `PhilosopherImpl`: State Loop in `run()`

```
1  for (; count < rounds && ! moved; count++) {
2     try {
3        switch(state) {
4        case HUNGRY:
5           self.eat();
6           state = State.THINKING;
7           break;
8        case THINKING:
9           self.think();
10          state = State.SLEEPING;
11          break;
12       case SLEEPING:
13          self.sleep();
14          state = State.HUNGRY;
15          break;
16       }
17    } catch (RemoteException e) { /* Elided for brevity. */ }
18 }
```

the cafés collectively produce enough food to sate all six philosophers, assuming the number of philosophers at each café matches the café's production rate.

In Listing 7.2, `rounds` and `count` are integers; `self` is a MAGE proxy to an instance of `PhilosopherImp`; and `moved` is a boolean, whose purpose is explained below. `PhilosopherImpl`'s constructor initializes each of these variables and binds a mobility attribute to `eat()`. When a philosophers calls its `eat()`, the mobility attribute bound to it decides at which café that philosopher should eat. In Sections 7.2.3 and 7.2.4, we describe and analyze the performance of various migration policies, realized as mobility attributes that we bind to `eat()`.

With the exception of `moved`, Listing 7.2 is a standard realization of the state loop in a classic formulation of dining philosophers. Here, the philosopher transitions between its three states of hungry, thinking, and sleeping until the count of those transitions exceeds `rounds`.

In MAGE, when the mobile object o moves from host s to host t, o is first cloned, then the new clone is marshaled and sent to t. The new clone at t executes the call that triggered

Listing 7.3: `eat()`

```
1  public void eat() throws RemoteException {
2     if (justArrived) // check whether to restart.
3        uponArrival();
4     else {
5        // The asymmetric philosopher's first and second
6        // chopsticks are switched.
7        getChopstick(first);
8        getChopstick(second);
9        int j = 0;
10       for (; j < helpings;) {
11          j += cafe.fillPlate(helpings - j); //blocks if no food
12       }
13       first.put(id);
14       second.put(id);
15    }
16 }
```

o to move and becomes the current version of *o*. MAGE discards the old clone that remains at *s*. `MageMobileObject` defines `moved`. It is initially false, but is set to true in the old clone when the new clone of a mobile object is sent to another host. Thus, `moved` distinguishes the old and new clones of a mobile object. In Listing 7.2, `moved` set to true causes the thread running in an old clone to exit the state loop and terminate.

Philosophers are active objects that move themselves. As we have noted, Java does not support strong mobility, so we need some way to restart a computation at the language level: we need to define a continuation using only the heap. When an active object cycles through a state machine encoded in a **`switch`**, we can store its current state in the heap. If each case in the **`switch`** is side-effect free before a movement-triggering method is called, then when we marshal, move, and unmarshal the object we can restore its state and start a new thread that returns to the method call that triggered the active object's movement. We must perform bookkeeping whenever an active mobile object arrives at a new host, notably we must start the mobile object's thread. All movement-triggering methods must detect arrival and perform such bookkeeping as necessary. We use these techniques to realize philosophers as simultaneously active and mobile objects.

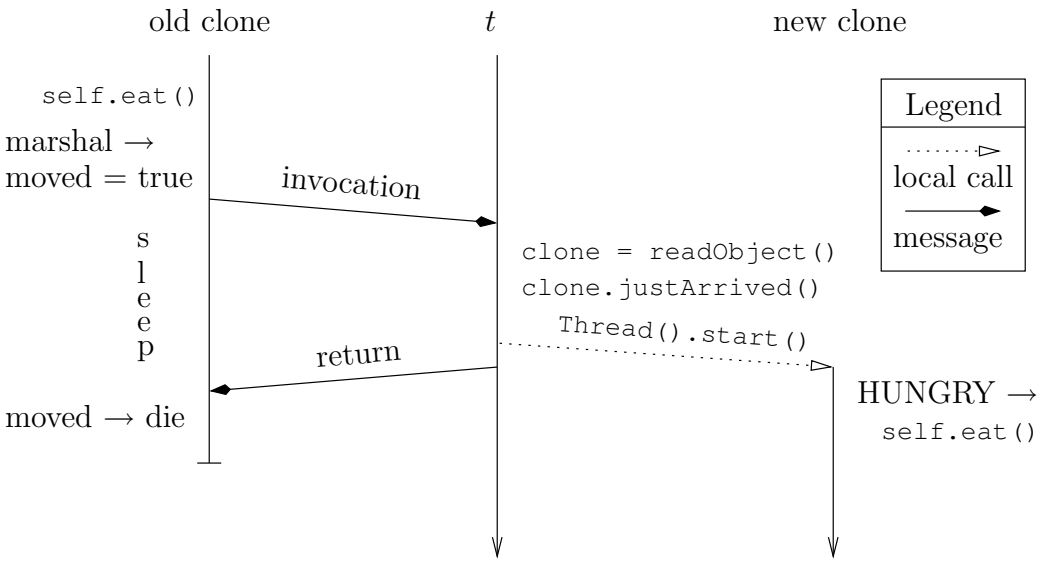


Figure 7.4: A Philosopher Moves to t

When a self-invocation causes a mobile object to move, the thread running in the old clone sleeps in the MAGE listener, as shown in Figure 7.4. Execution in the new clone sends a return to the listener and awakens the old clone's thread. The old clone's thread must be side-effect free. For this reason, a method whose self-invocation causes movement must be void. The old clone's thread can, of course, write to the old clone, as that clone is simply discarded.

Listing 7.3 defines a peripatetic dining philosopher's movement-triggering method, its `eat()` method. The variables `first` and `second` are fields that contain proxies to a philosopher's chopsticks. In their constructors, philosophers bind these fields to `CODa`. The `helpings` field determines how much a philosopher eats when it is hungry. It defaults to one in the evaluations that follow. This `eat()` differs from its classic formulation in dining philosophers in two ways: the if-else and its `getChopstick` method.

The `eat()` method's if-else handles bookkeeping. When a philosopher moves and its new clone arrives at host, `readObject()` in Listing 7.4 unmarshals in it and sets `justArrived` to true. Thus, a movement-triggering call executes `justArrived` branch in the body of `eat()` and chains to `uponArrival()` in Listing 7.5, which first clears `justArrived`, performs the bookkeeping, then starts a new thread for the philosopher.

Listing 7.4: `readObject()`

```
1 private void readObject(ObjectInputStream in)
2    throws IOException, ClassNotFoundException
3 {
4    in.defaultReadObject();
5    justArrived = true;
6    moves++;
7 }
```

Listing 7.5: `uponArrival()`

```
1 private void uponArrival() throws RemoteException {
2    justArrived = false;
3    try {
4       init(); // Get proxy and bind mobatt.
5    } catch (RemoteException e) { /* Elided for brevity. */ }
6
7    new Thread(this).start();
8 }
```

The movement-triggering call then returns and awakens the old clone's thread, which sets the old clone's state to `THINKING` before executing the loop conditional. Since the old clone's `moved` boolean is true, it dies and the fact that it changed the old clone's state is irrelevant. The new clone's thread starts out with its state set to `HUNGRY`, so it immediately executes `eat()`. This execution applies the philosopher's mobility attribute which could cause the philosopher to move again, repeating this process. To prevent starvation, all the mobility attributes in this evaluation also have a `justArrived` field, and return the local host when it is true, thus allowing the philosopher to execute `eat()` at least once after every move. If there is no food, the philosopher blocks in a queue until the café produces more food.

There are two ways to immobilize a mobile object in MAGE. The first is an internal move lock. A mover holds this lock while it waits for inplace invocations currently executing in a mobile object to complete before it moves the object. To prevent the starvation of movers, this move lock prevents new inplace invocations. Only a single mover can hold this lock at a time.

Listing 7.6: `ChopstickImpl` Methods

```
1  public synchronized void put(int id) throws RemoteException {
2     if (owner != id) {
3        return;
4     }
5     mobilize(); //allow movement
6     owner = -1;
7     notifyAll();
8  }
9
10 public synchronized void get(int id) throws RemoteException {
11    while (owner != -1)
12       try {
13          wait();
14       } catch (InterruptedException e) {}
15    immobilize(); //Prevent movement, until put() called
16    owner = id;
17 }
18
19 public synchronized void waitUntilFree() throws RemoteException
20 {
21    while (owner != -1)
22       try {
23          wait();
24       } catch (InterruptedException e) {}
25 }
```

The second mechanism is user-visible. Two methods defined on `MageMobileObject` realise this mechanism: programmers call `immobilize` to lock a mobile object to its current host and `mobilize` to free it. In peripetatic dining philosophers, the chopsticks represent mobile resources that the philosophers need in order to eat. We have defined need to mean that the chopsticks must be local when a philosopher eats, thus the binding of `cod` to them. In the absence of user-controlled immobilization, while a philosopher was waiting for its second chopstick, its first chopstick could be moved away by the philosopher that shares that chopstick with it.

Listing 7.6 illustrates the use of these two methods. In MAGE, a mobile object moves before the method triggering that movement executes. Thus, a philosopher does not check whether its neighbor owns the chopstick until after the chopstick is collocated with it.

Listing 7.7: `getChopstick(Chopstick chopstick)`

```
1  protected void getChopstick(Chopstick chopstick)
2     throws RemoteException
3  {
4     int i = 1;
5     for (; i < 100; i++) { //Tolerate acquisition race.
6        try {
7           chopstick.get(id);
8           break;
9        } catch (RemoteException e) {
10          if (e instanceof ImmobilizedException)
11          {
12             chopstick.rebind(cle);
13             chopstick.waitUntilFree();
14             chopstick.rebind(cod);
15          }
16          else
17             throw e;
18       }
19    }
20    if (i >= 100) { //Lost acquisition race too many times.
21       System.exit(1);
22    }
23 }
```

Both philosophers could be collocated, so the method must be synchronized. The owner immobilizes the chopstick so that both chopsticks are local when it eats.

Listing 7.7 defines a philosopher's `getChopstick()` method. This method wraps the classical acquisition of a chopstick in both a **`for`** loop and a **`try-catch`** block. The **`try-catch`** block handles the case where the chopstick has been immobilized. Rather than spin, we first bind an instance of a current location evaluation mobility attribute so that we can execute `waitUntilFree()` in place. When the thread waiting in `waitUntilFree()` awakens, it sends a return to the thread waiting in `getChopstick()`, which must rebind `cod`, before again making the remote call to the chopstick's `get` method. The latency of this work and the fact that the philosopher that just freed the chopstick is local to the chopstick means that philosopher that was waiting in `waitUntilFree()` often loses the race. The **`for`** loop tolerates this race.

7.2.2 Migration Policies

To support some of the mobility attribute we describe below, our implementation of peripatetic dining philosophers populates the MAGE resource manager with each café's production rate r, number of resident philosophers, and available food m. Realized as mobility attributes, the migration policies are

Cooperative, CooperativeLocal These two policies seek to cooperatively move the philosophers to match the consumption rate of the philosophers at a café with that café's production rate. When |Local Philosophers| $= n > r$, each philosopher moves to another café with probability $\frac{n-r}{n}$. Thus, the expected number of moves is precisely the number of philosophers in excess of the café's rate of production. These two attribute differ in that Cooperative sends messages to discover a target café where $r > n$, while CooperativeLocal eschews messages and randomly picks a target.

Gourmand This policy ranks the cafés and causes a philosopher to move to its highest ranked café that has food.

Hermit This policy causes a philosopher to seek a café with the fewest other philosophers on it.

MostFood This policy greedily moves a philosopher to the café with the most available food.

HighestProduction This policy moves a philosopher to the café with the highest rate of production.

RandomWaitJaunt This policy ignores 1 to 10 calls to `eat()` chosen uniformly at random, before choosing a café uniformly at random.

Listings 7.8, 7.9, and 7.10 present the "MostFood" policy's concrete implementation as a mobility attribute.

In Listing 7.8, `OneMethodMobAttA`, which `MostX` extends, is an attribute that MAGE provides in its `ml` package for use as the base class of all attributes that bind to a single

Listing 7.8: `MostX` Attribute

```
1  package util;
2
3  import java.io.IOException;...
4
5  public class MostX extends OneMethodMobAttA {
6
7     private static final long serialVersionUID = 1L;
8     String resource;
9     private boolean justArrived;
10    protected transient ResourceManagerServer rm;
11
12    public MostX(Class iface, String method,
13          Class[] params, String resource)
14       throws SecurityException, NoSuchMethodException
15    {
16       super(iface, method, params);
17       this.resource = ":" + resource;
18       rm = ResourceManagerServer.getResourceManagerServer();
19    }
20
21    @Override public Set<String> targets()
22       throws TargetException
23    { /* See Listing 7.9 */ }
24
25    private void readObject(ObjectInputStream in)
26       throws IOException, ClassNotFoundException
27    {
28       in.defaultReadObject();
29       justArrived = true;
30       rm = ResourceManagerServer.getResourceManagerServer();
31    }
32 }
```

method. `MostX` assumes that the MAGE resource manager is populated with keys, whose format is `host:resource`, mapped to float values. The `X` in its name abstracts the resource name. Its `rm` field is a handle to the local resource manager. As such, it is transient; whenever this attribute is unmarshaled, `readObject` re-initializes it.

As described above in Section 7.2.1, `justArrived` avoids starvation by forcing philosopher to attempt to eat after each move. Set in `readObject`, it is checked in the `targets`

Listing 7.9: `MostX targets()`

```
1  @Override public Set<String> targets() throws TargetException {
2     Set<String> ret = new HashSet<String>();
3     ret.add(MageRegistryServer.getLocalHost());
4     if ( ! justArrived) {
5        Map<String,Object> rmap = rm.getMap();
6        boolean hit = false;
7        Object value;
8        float max = Float.MIN_VALUE, current;
9        for (String key : rmap.keySet()) {
10          if (key.endsWith(resource)) {
11             hit = true;
12             value = rmap.get(key);
13             if (value == null) break; // handle startup latency
14             if (value instanceof Float)
15                current = ((Float)value).floatValue();
16             else
17                current = ((Integer)value).intValue();
18             if (current > max) {
19                max = current;
20                ret.clear();
21                ret.add(key.split(":")[0]);
22             }
23             else if (current == max)
24                ret.add(key.split(":")[0]);
25          }
26       }
27       if ( ! hit)
28          throw new TargetException(
29             "The resource " + resource.split(":")[1]
30             + " not defined."
31          );
32    }
33    else
34       justArrived = false;
35
36    return ret;
37 }
```

Listing 7.10: `MostFood` Attribute

```
1  package util;
2
3  import java.util.HashSet;...
4
5  public class MostFood extends MostX {
6
7     private static final long serialVersionUID = 1L;
8     protected int interval;
9     protected long last;
10
11    public MostFood(Class iface, String method,
12          Class[] params, int interval)
13       throws SecurityException, NoSuchMethodException
14    {
15       super(iface, method, params, "food");
16       this.interval = interval;
17       last = 0;
18    }
19
20    @Override public Set<String> targets()
21       throws TargetException
22    {
23       long now = System.currentTimeMillis();
24       if (now - last > interval) {
25          last = now;
26          return super.targets();
27       }
28       else {
29          Set<String> ret = new HashSet<String>();
30          ret.add(MageRegistryServer.getLocalHost());
31          return ret;
32       }
33    }
34 }
```

method of `MostX` in Listing 7.9. When `justArrived` is true, `targets()` returns a singleton set containing only the philosopher's current host. Otherwise, it loops through all keys whose suffix matches the `resource` field, returning the set of hosts that share a maximum value for that resource.

In Listing 7.10, `MostFood`'s `targets()` is quite simple and compact. It simply checks if sufficient time, as defined by the sampling interval (ms) passed into its constructor has elapsed. If not, it returns the singleton set of the host on which it is executing. Otherwise, it calls the `targets()` method of its superclass, `MostX`.

Total Physical Source Lines of Code (SLOC)	=	545
Development Effort Estimate, Person-Years (Person-Months)	=	0.11 (1.27)
(Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))		
Schedule Estimate, Years (Months)	=	0.23 (2.74)
(Basic COCOMO model, Months = 2.5 * (person-months**0.38))		
Estimated Average Number of Developers (Effort/Schedule)	=	0.46
Total Estimated Cost to Develop	=	$14,284
(average salary = $56,286/year, overhead = 2.40)		

Table 7.3: Aggregate Policy Metrics

David A. Wheeler's 'SLOCCount' tool [106] generated the data in Table 7.3 and Figure 7.5. This tool strips comments and then counts the remaining, physical, not logical, source lines of code (SLOC). Further, we have used this tool to count each attribute's entire class file, not just their target methods.

COCOMO is a widely used software cost estimation model [15]. We report it here to translate the SLOC results into dollars, a more meaningful unit of measure. Realized as mobility attribute under MAGE, migration policies are isolated and small: developers working on distributed systems that employ mobility are likely to save money if they use MAGE.

Figure 7.5 presents the SLOC of the application attributes — attributes written to realize migration policies for the peripatetic dining philosophers application — not the attributes that MAGE provides in its `ml` package. Like `MostX`, the `FewestX` and `NoX` classes factor code used in subsets of the policies. In particular, `Hermit` extends `FewestX` and `Gourmand` delegates to a `NoX` instance.

These metrics demonstrate the concision of the policies that MAGE can express: MAGE allowed us to express a number of interesting policies, well-suited for exploring layout in the setting of peripatetic dining philosophers with minimal effort.

7.2.3 Fixed Production

In this section, we compare migration policies against each other and three immobile layouts when the café's food production is fixed as defined in Figure 7.3b. The immobile layouts motivate mobility by quantifying the benefit in terms of time-to-completion and

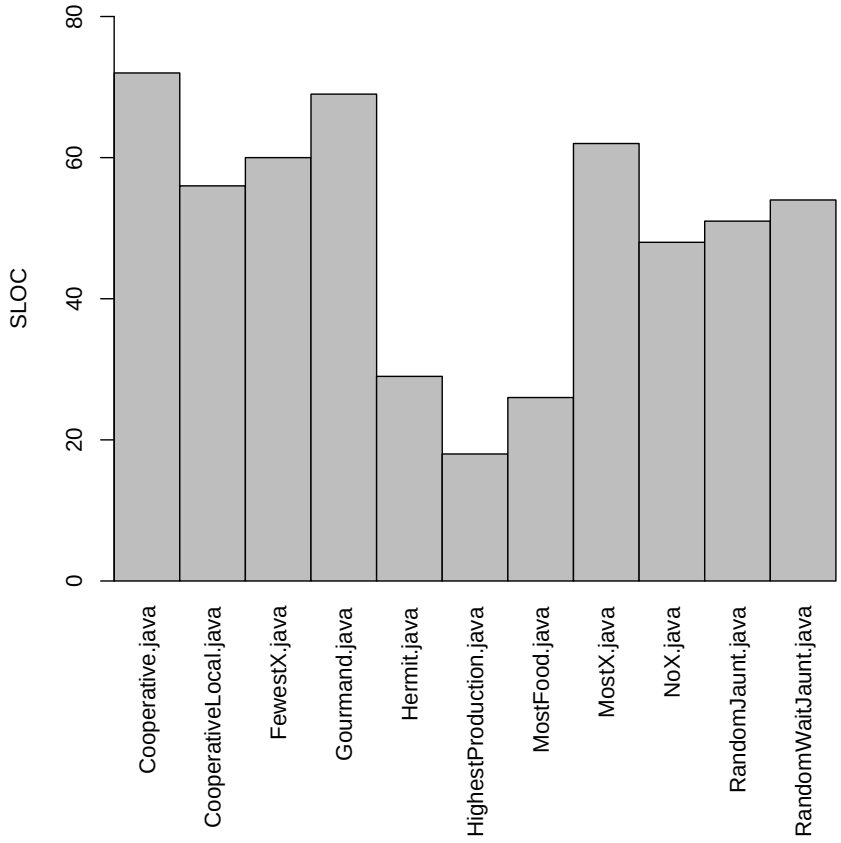


Figure 7.5: SLOC per Policy

resource utilization that an application, here peripatetic dining philosophers, can realize by dynamically changing its layout. The immobile layouts are

ImmobileBestCase One philosopher starts at the café whose production is 1 meal/s, 2 on the café whose production is 2 meals/s, and 3 on the café whose production is 3 meals/s.

ImmobileWorstCase All 6 philosophers start on the café whose production is 1 meal/s.

ImmobileRandom Each philosopher's starting location is chosen uniformly, at random.

The best case layout requires extrinsic information, here the production rate of each café. For our evaluation, the worst case layout, which places all philosophers at the slowest

producing café, is unlikely, at $\frac{1}{3}^6 = \frac{1}{729}$. We include it as a point of reference. The random layout is a conservative approximation of an arbitrary program's layout in the absence of extrinsic information.

The collections of migration policies we evaluate in Figures 7.6–7.8 include all Cooperative, all CooperativeLocal, all Greedy with various sampling rates, all MaxProduction, and all RandomWaitJaunt. MinimizeMoves minimizes philosopher moves using the same extrinsic information that underlies the static best layout to bind an instance of the `Hermit`, two instances of `Gourmand`, and three instances `MaxProduction` attributes to the six philosophers. RandomPolicy binds a policy from the above policies to each philosopher uniformly at random.

We compare these migration policies and static layouts along four axis — time to completion, spoiled food/wasted food, philosopher moves, and chopstick moves.

Each bar chart in this section reports the average of 10 runs. The philosophers make 150 transitions from `HUNGRY` to `THINKING` to `SLEEPING` in their three state state machine, so they call their eat method 50 times. There are six philosophers, who each eat 1 meal when they are hungry and three cafés that, in aggregate, produce 6 meal/s. Thus, ignoring network latency and execution time, the idealized minimum time-to-completion is 50s. When evaluating the migration policies, we start the philosophers out at a café chosen uniformly at random.

The messaging overhead of discovering an optimal target shows up in the difference in time-to-completion of Cooperative and CooperativeLocal in Figure 7.6. As expected, local cooperative causes philosophers to move slightly more often, with its concomitant impact on chopstick moves, as shown in Figures 7.9 and 7.10. The box plot in Figure 7.7 demonstrates that, for most policies, the variation is quite low.

Both random policies — RandomPolicy and RandomWaitJaunt illustrate the utility of mobility. Random policy conservatively models the effect of mobility when one lacks extrinsic information about either the application's behavior or resource distribution and production. The random policy dramatically demonstrates the utility of mobility — a random collection

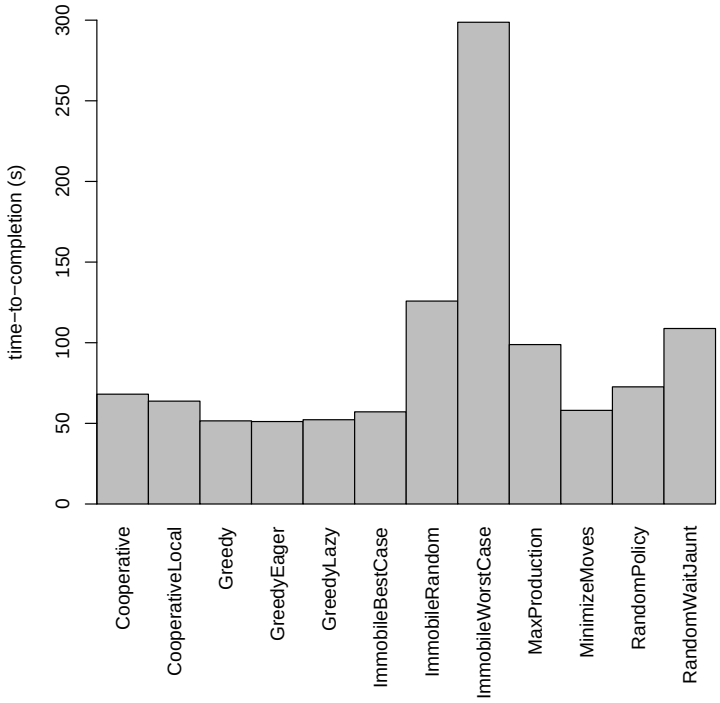


Figure 7.6: Fixed Cooks: Average Time-to-Completion

of migration policies finishes in 57% of the time and 30% of the waste when compared to the random, immobile layout. Even though RandomWaitJaunt's decisions are all random, it outperforms immobile random layout in terms of time-to-completion and wasted food since it distributes the philosophers more evenly across the cafés temporally than immobile random.

MaxProduction sacrifices time-to-completion, since it concentrates all the philosophers on the highest producing café, in exchange for fewer moves.

The three greedy policies differ in the rate at which they sample the resource manager for each café's available food. The eager version's interval is 50ms; greedy's is 200ms; and the lazy version's is 500ms. Note that café's update their food every 50ms, so sampling faster than 50ms would waste resources in this application.

Figure 7.6 demonstrates that the three greedy policies are very fast: they are the three fastest policies, finishing very close to the optimal time of 50s. The greedy policies also

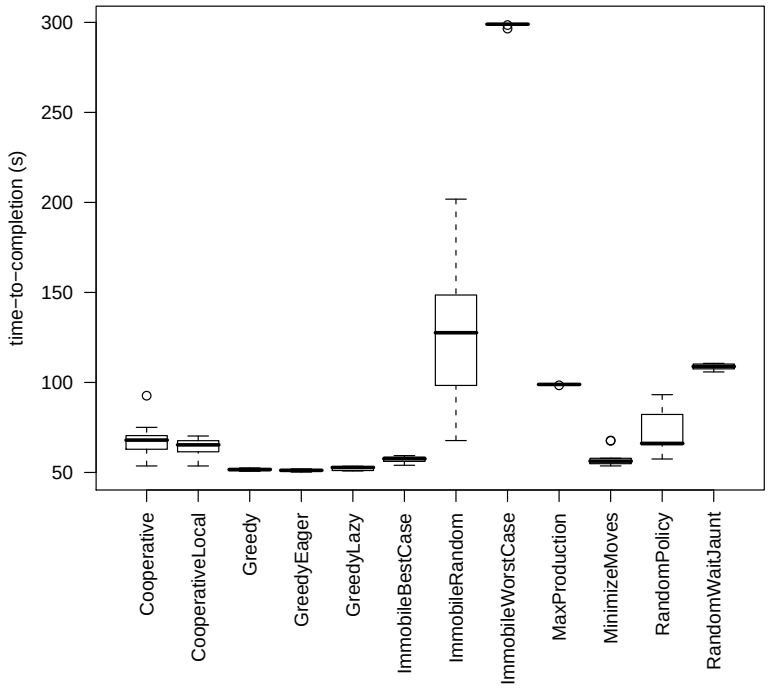


Figure 7.7: Fixed Cooks: Variation in Time-to-Completion

minimize food wastage, as shown in Figure 7.8. In particular, the three greedy policies perform better than the best case layout of immobile philosophers. At first, this seems remarkable. However, the three cafés' production of meals is not synchronized. The last greedy philosopher to finish does so as soon as the last food he needs is produced anywhere. The last immobile philosopher must wait until his café produces his final meal.

To achieve these numbers, the greedy policies move the philosophers around frequently, as Figure 7.9 makes clear. Under the greedy policies, moreover, the philosophers all move en masse from their current café to the café with the most available food, dragging their chopstick along with them. This is the key to why the greedy policy performs so well — most chopstick contention is local. In an evaluation instrumented to track chopstick acquisition time, the average time to acquire a chopstick under the cooperative policy was 2.5s, while that of greedy was 1.3s. Thus, the greedy policy's chopstick acquisition time is 0.53% or slightly great than $\frac{1}{2}$ the acquisition time of the cooperative policy.

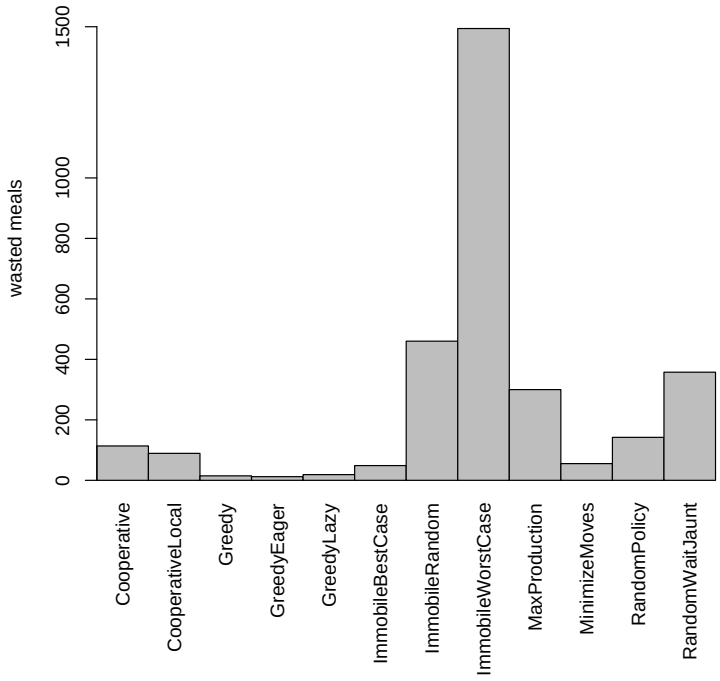


Figure 7.8: Fixed Cooks: Average Number of Uneaten Meals

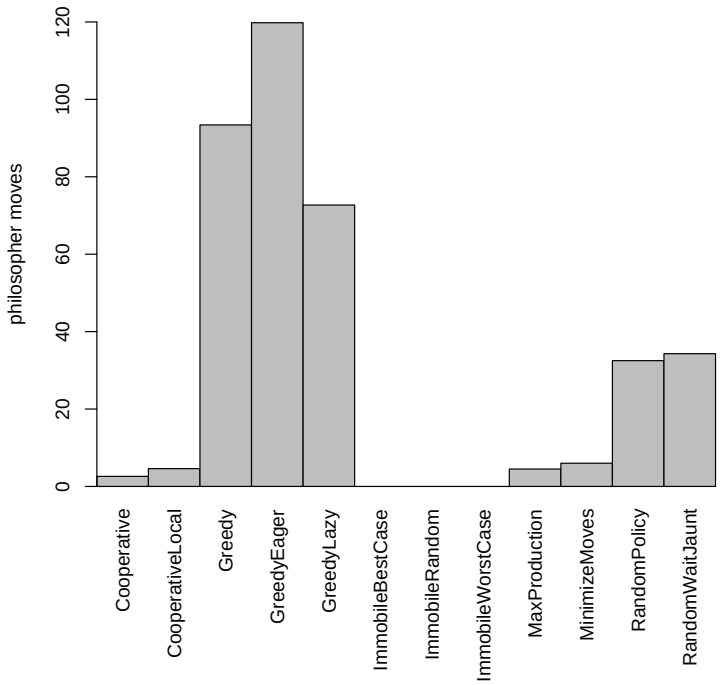


Figure 7.9: Fixed Cooks: Average Number of Philosopher Moves

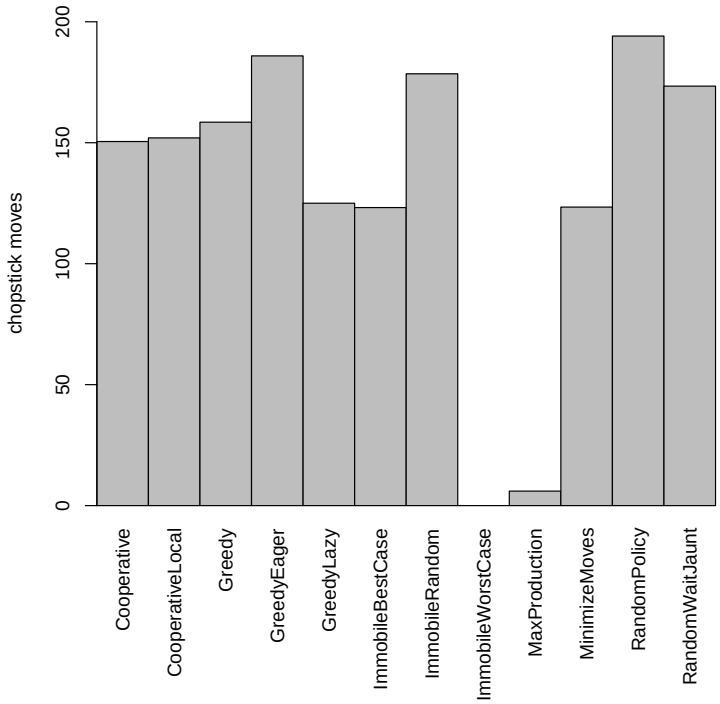


Figure 7.10: Fixed Cooks: Average Number of Chopstick Moves

7.2.4 Changing Production

When the distribution of resources is fixed albeit unknown, mobility serves only to allow an application to recover from a poor initial deployment of its components. When resources are themselves mobile and changing, an application composed of mobile objects can adapt as the distribution of resources changes. In this section, we evaluate policies in the context of changing resources. Specifically, each café's rate of food production starts out randomly distributed, then changes every minute.

We change runs from 150 to 1500 so that there are more opportunities for the application to react to changes in food production at the cafés. 1500 transitions means that each philosopher needs to eat 500 meals. Since there are 6 philosophers, the earliest the experiment can end is when the cafés produce 3000 meals. To ease the comparison of runs, we maintain the invariant that the cafés always produce 6 meals/s so that requisite food is produced in very nearly the same amount of time. Given this constraint, the optimal time-to-completion is $3000\text{meal}/6\text{meals/s} = 500\text{s} = 8.\bar{3}\text{m}$. Each café changes cooks, *i.e.* its meal production rate, every minute, for a grand total of 8 times during each run.

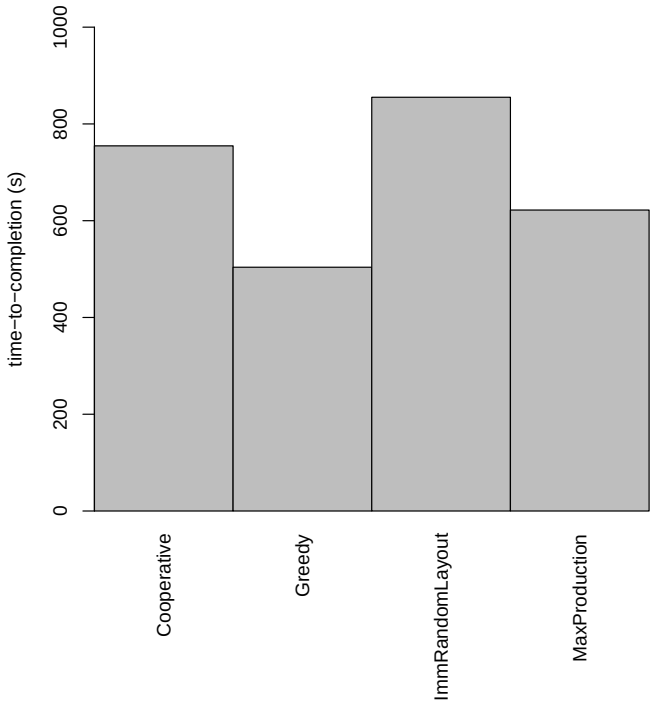


Figure 7.11: Changing Cooks: Average Time-to-Completion

The policies we evaluate are Cooperative, Greedy, Immobile Random Layout, and Maximum Production. When production is changing, immobile best case and worst case random layout are not well-defined as a layout could change from best case to worst case as the distribution of production rates changes.

In Figure 7.11, Greedy finishes first, followed by MaxProduction. The cost advantage of local chopstick acquisition dominates Cooperative's better layout of philosophers to cafés. Figure 7.12 demonstrates Greedy's excellence in quickly using all available food. Figure 7.13 depicts what the Greedy policy expends to achieve its time-to-completion and wasted meals numbers: it reconfigures its philosophers frequently.

MaxProduction is the outlier in Figure 7.14. It concentrates all the philosophers at the café that currently has the highest production, so it moves the philosopher, dragging their chopsticks with them, only 9 times. Greedy is high in spite of the fact that it also concentrates the philosophers at a single café, because, as already noted, it frequently moves

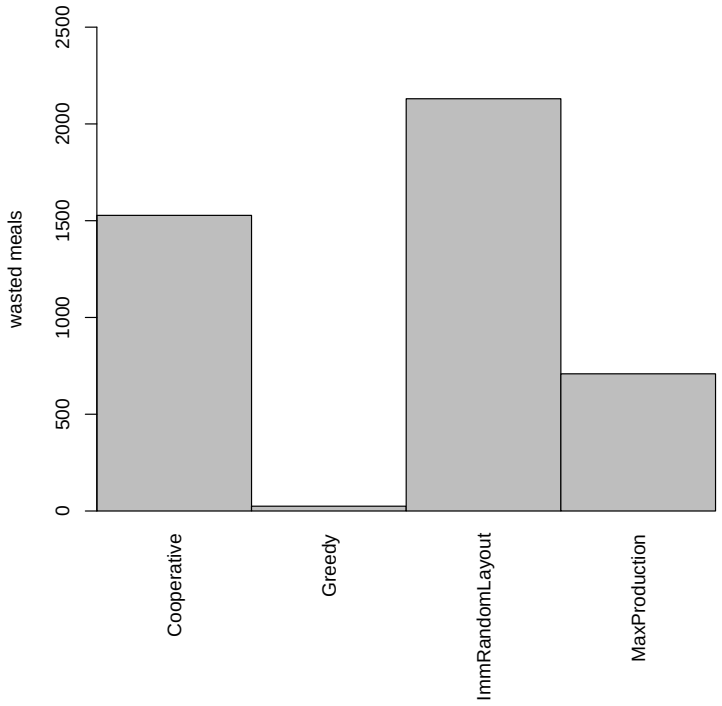


Figure 7.12: Changing Cooks: Average Number of Uneaten Meals

the philosophers, thus dragging the chopsticks along.

7.2.5 Future Work

Our evaluation of the proposed migration policies has been descriptive. Game theory provides tools that can answer prescriptive questions such as "which collection of policies should one use?" We can model philosopher layout, and more generally mobile object layout, as a game. We could define the payoff function that describes different outcomes in terms of various combinations of the metrics we have analyzed, such as time-to-completion. A migration policy would then be the strategy that a philosopher follows. We could then apply game theoretic solution concepts, such as Nash or dominant strategy equilibria, to find the corresponding set of strategies. Further criteria from economics, such as welfare maximization, can also be applied.

In the above evaluation, the cost of philosopher migration is fixed and small. An

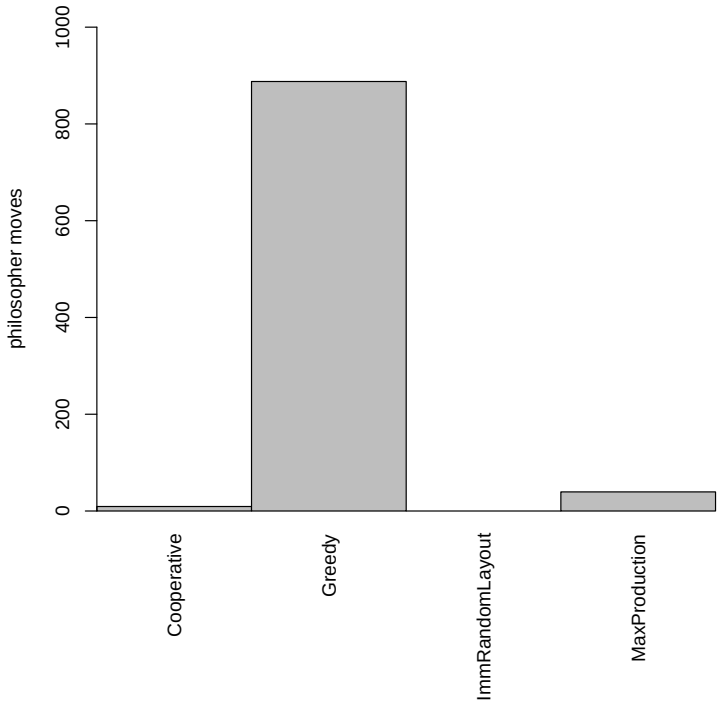


Figure 7.13: Changing Cooks: Average Number of Philosopher Moves

interesting evaluation would vary this cost to determine, for instance, the point at which Cooperative finishes before Greedy.

Two philosophers share each chopstick. Chopstick acquisition is more expensive when the philosophers that share a chopstick are not collocated. Let philosophers be numbered [1..6]. Consider the partition $\{1\}, \{3, 5\}, \{2, 4, 6\}$, each assigned to a different café. This particular layout maximizes the number of edges in the dining philosopher ring that cross the partitions. It would be interesting to study policies that are aware of the philosopher ring and seek to minimize these crossings. When chopsticks are fungible resources, such minimization might be achieved by dynamically changing the dining philosopher ring, *i.e.* by exchanging a pair of chopsticks between a pair of philosophers. To our knowledge, exchanging chopsticks has not been previously studied: the authors of the evolving philosopher problem considered only the birth and death of a philosopher and the merging of two communities (rings) of philosophers [53].

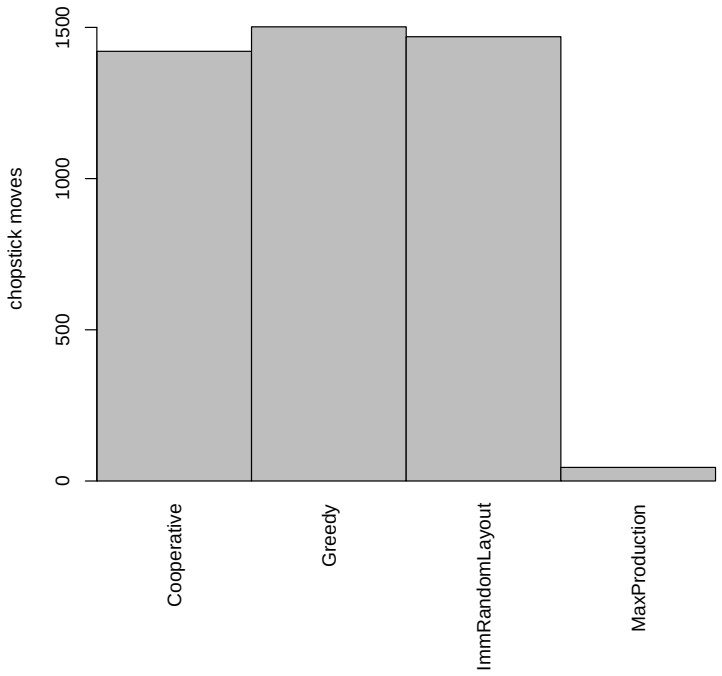


Figure 7.14: Changing Cooks: Average Number of Chopstick Moves

Finally, it may be interesting to investigate peripatetic dining philosophers as a linear programming problem.

7.3 Summary

In this chapter, we have quantified MAGE's invocation and marshaling overhead using benchmarks. We have presented peripatetic dining philosophers, a variant of dining philosophers designed for a mobile context. Featuring both mobility of philosophers and chopsticks, this demanding test demonstrates the robustness of the MAGE implementation. We examined various migration policies and showed how concisely MAGE can express these policies.

Chapter 8

Related Work

There is nothing new under the sun.

Ecclesiastes, 1:9 NIV

Computing environments change. Machines fail and new ones are added. Gigabit switches are installed and printers decommissioned. New applications and operating systems are installed. Mobile devices, like cellphones, change location and can access a different set of resources than they could in their previous location. Network and CPU load varies. Often, administrators must handle these changes. The challenge is to build systems and applications that reliably handle change, without human intervention.

This challenge is seminal. Researchers have worked on it since the dawn of the computer age. We take some of this work for granted, like adaptive locks or the fact that file systems test blocks and, once they identify a bad block, they recover the data and avoid that bad block in the future. In this vast space of work, Figure 8.1 focuses on two overlapping subsets, systems that, in response to their environment, change their behavior or change their layout, the mapping of their subcomponents onto execution engines. To differentiate behavioral and layout adaptation, we present two canonical examples:

Consider a client-server multimedia system that streams data from the server to the client for playback. This system can encode the stream in two ways: 1) a bandwidth expensive

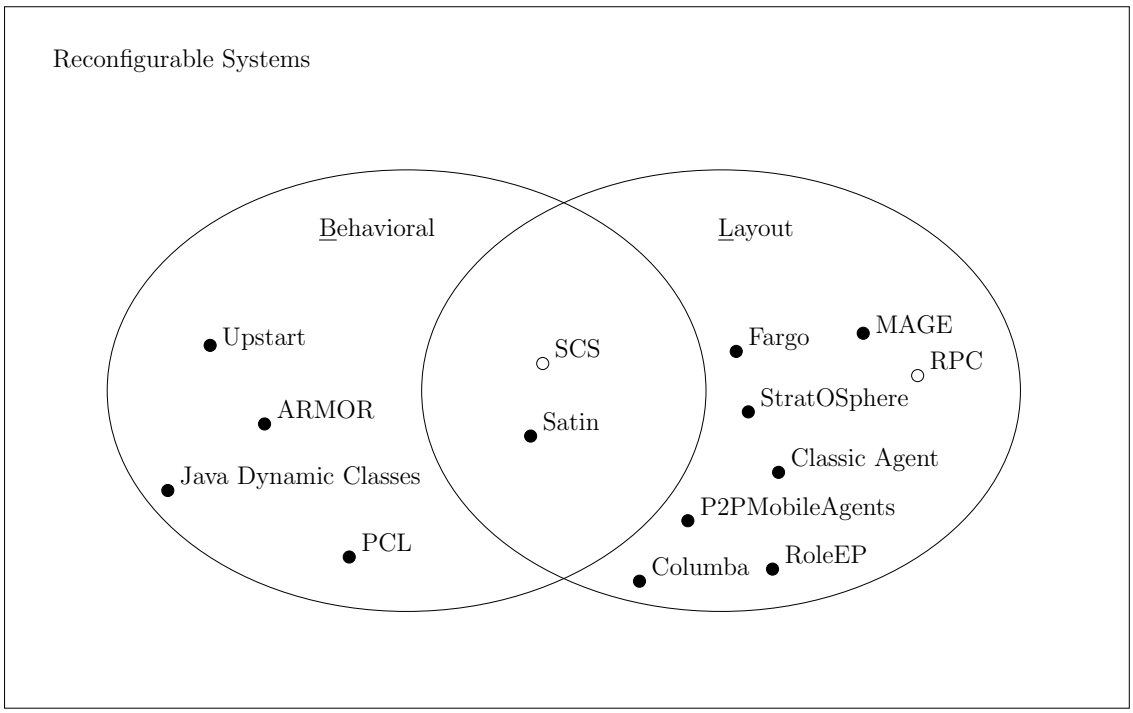


Figure 8.1: Reconfigurable Systems

but high-fidelity encoding or 2) a light-weight but lossy encoding. If this system switches between these two encodings depending on network load, the system exhibits *behavioral* adaptation [28, §3.1]. A cluster whose scheduler moves running components around to maximize its utilization, exemplifies layout adaptation.

Behavior adaptation changes the binding of a method or service name to an implementing algorithm. Code mobility is "the capability to reconfigure dynamically, at runtime, the binding between the software components of the application and their physical location within a computer network" [16, Introduction]. When behavior is sensitive to layout, behavioral and layout adaptation converge. REV's motivating example illustrates this convergence: a mail server performs custom filtering using filters sent by its clients[1].

The goal of behavioral adaptation is to replace running code automatically, either to limit service disruption during an upgrade or in response to the environment, as described above. Difficulties include handling the reappearance of systems that did not receive an

[1] Chapter 4 opens with this example.

update because they were unavailable, and correctly transitioning threads running the old behavior to the new behavior [64, 4, 107, 28]. As these systems are not closely related to MAGE, we do not discuss them further here.

Adaptation can either be static, via deployment or initialization, or dynamic. Dynamic subsumes static, as a dynamic system can reconfigure itself at startup, thus simulating static adaptation. We use shading to denote adaptation time offered by a project in Figure 8.1. Hollow circles denote systems that adapt statically. Solid circles denotes systems that adapt dynamically.

The self-configuring systems (SCS) project is an example of a system that statically adapts itself autonomously [45]. Given a recipe that selects algorithms (behavior) and layout based on resource distribution and availability, SCS queries its environment to generate a site-specific configuration. Because it adapts both behavior and layout, SCS is in the intersection.

Classically, layout has been manual and static, restricted to deployment. For instance, RPC-based systems have assumed static distribution of components and their definitions. Java's RMI [66], CORBA [92], and COM/DCOM [27] exemplify such RPC-based distributed system infrastructures.

The idea of supporting dynamic layout, *i.e.* program mobility, is not new and has appeared in various forms in distributed operating system [7, 26, 61] and programming language [48, 37] research. Broadly, this research has explored systems that offer ever greater degrees of mobility, progressing from the data migration inherent to RPC [13] to the explosion of interest in MA [99] that began in 1995. The vast majority of mobile agent systems support mobility through explicit calls to a `move` command. As a result, the programmer must intermingle the application's layout, or per component migration policies, with the code that implements the application itself.

Not least for this reason, mobile programs are more complex to write and debug than distributed programs that rely on a static layout of their components. As a result, a number of programming models that abstract dynamic layout have been proposed. MAGE is one such programming model. In the rest of the chapter, we compare and contrast MAGE with

its conceptual siblings.

8.1 Classic Agent

Examples of early work on mobility of programs (and objects) through a language's runtime system are Emerald [48], Hermes [14], DOWL [2], and COOL [37]. In particular, Emerald introduced the ability to "*locate* an object, *move* an object to another node, and *fix* an object at a particular node." By moving objects, these early systems achieved a weak, heap-based form of mobility. In the 1990s, a second wave of mobile code languages burst onto the scene, including Telescript [108], AgentTCL [52], Aglet [58], Mole [96], Ara [79], Ajanta [101] and Sumatra [85]. We can classify [50] these systems into those that move execution state as well as program code (strong mobility), and those that move only code (weak mobility). Examples of systems that support both are Sumatra and Telescript. The latter impose constraints on which components can move when to achieve mobility.

Since it is expensive to access and move a thread's stack safely and the JVM does not provide access to the CPU's register file, MAGE currently implements a form of weak mobility: In general, it waits for all threads to drain from a mobile object before moving it. MAGE *can* move active objects, as in the peripatetic dining philosopher case, but only if the active object cooperates by providing its own bus stops, execution points in which the thread is in a known state and thus suitable for movement.

These languages provide some form of a `move` command that programmers can use to change the location of a program's components. They stand in the same relation to programming models like MAGE and its siblings in the same way that assembly language undergirds higher-level languages, their `move` command underlies higher-level movement constructs in the same way that synchronization primitives can be reduced to semaphores.

Listing 8.1 depicts the code an invoking thread would need to execute to realize COD in a classic agent language. The invoker must discover its own location, collocate `a` with itself, before executing the desired operation.

Listing 8.1: COD in Classic Agent Language

```
1 local = getLocalHost();
2 a.move(local);
3 result = a.operation();
```

8.2 FarGo

The FarGo programming model [42, 44, 43, 1] calls its components *complets* and defines them to be the transitive closure of the heap references of an object. It is unclear how a *complet* differs from Java's serial representation of an object, which is also the transitive closure of heap references [67]. FarGo provides programmers three ways to control component mobility. The first is an explicit call to `move`. FarGo supports two implicit mobility mechanisms — events and relocation semantics attached to inter-*complet* links.

Inter-*complet* references are essentially stubs in RMI terms. FarGo supports five default relocation policies in a set of *complet* reference classes — Link, Pull, Duplicate, Stamp, and Bi-directional Pull. Programmers can implicitly control component mobility by attaching instances of these relocation policy classes to inter-component references. For example, if a component `A` is interconnected to component `B` by the Pull relocator (Pull is directional), then when `A` moves, `B` follows it. The `A` component could have been explicitly moved, moved by an event, or moved because of the semantics of an inter-component link applied to it. A FarGo programmer can use these links cause groups of components to move in unison.

FarGo allows programmers and administers to write *complet* migration policies using *complet* references, a Java event API, an Event-Action scripting language, and a graphical layout tool. The graphical tool relies on the Java event API and script mechanisms, so we will not discuss it further. Both the Java event API and script mechanisms allow FarGo programmers to register interest in events, such as the departure or arrival of a *complet*, or the startup or shutdown of an execution environment. When an event of interest occurs, a programmer supplied migration policy executes. The Java event API intermingles component migration policy with application code; the scripting language separates the two. Events and *complet* references allow programmers to express migration policies in two different ways that

can interact: the migration of a component caused by an event can trigger the migration of other components due to *complet* references and vice versa. FarGo programmers must take care to avoid unintended interactions between these two policy mechanisms.

Unlike FarGo, MAGE offers a single mechanism, the mobility attribute, for implicitly controlling component mobility and expressing migration policies. Mobility attributes and FarGo's inter-*complet* references are both first class entities in the sense that they can be passed into functions and modified within Java, their implementation language. Both FarGo and MAGE allow the dynamic adaptation of migration policies through the binding and rebinding of their *complet* links and mobility attributes to components at runtime. FarGo's event-action scripts, while not first class, can also be dynamically bound to *complets* in FarGo. MAGE allows programmers to define their own policies by subclassing a default mobility attribute. FarGo's *complet* link classes are not intended to be extended by programmers. MAGE's mobility attributes can be composed (Section 3.5); FarGo makes no provision for the composition of either its *complet* links or its event-action scripts.

FarGo's *complet* reference mobility allows programmers to ensure that tightly coupled *complets* migrate and thus remain together. In other words, FarGo can move components independent of message exchange. Unilateral movement of a set of components is particularly important when moving components to and from hosts that are frequently disconnected, such as laptops [80]. Components can form transitory working sets in the course of a program's computation. In general, these working sets are difficult to statically discover. MAGE's invocation based mechanism lets a program's execution determine how the program's components move based on messages actually sent, rather than a programmer's static, and possibly incorrect notion, of component interaction. FarGo's event-based mechanism offers a natural way for a distributed application to react to the a machine going offline for maintenance, since it has only to post the event. In MAGE, the components would move off the host only as they were invoked, so a programmer would have to manually invoke each object on a host to simulate a host shutdown event.

8.3 StratOSphere

Inspired by a taxonomy of distributed execution scenarios proposed by Semeczko et al. [89], the StratOSphere project [118, 120, 119] maps these scenarios to computation paradigms, which it represents as messages. StratOSphere calls those scenarios that move only data to the target execution environment RPC, or Client/Server (CS) and represents CS as an exchange of messages. StratOSphere messages implement an interface that declares dispatch and send methods. To realizes COD, StratOSphere first sends a `MobileObjectMessage` message to fetch code from a remote system, before calling dispatch on the return message. StratOSphere's REV message, `RemoteEvaluationMessage`, carries local components to a remote server that unmarshals the message and calls its dispatch method.

The StratOSphere authors noted that many scenarios could not be represented by their CS, COD and REV messages and proposed a new paradigm, which they called Remote Code Evaluation (RCE) to capture these scenarios. An RCE computation is any computation that, from the point of view of the invoker, requires components from the local execution environment and other execution environments remote to the target execution environment. StratOSphere represents this paradigm as an RCE message (`RemoteExecutionMessage`) that conveys the local components to the remote target execution environment, and then gathers the remaining components from the nontarget remote execution environments before execution proceeds at the target.

StratOSphere makes these messages, and thus the computation paradigms they represent, available to programmers through a class hierarchy. Because its messages are passive, StratOSphere introduces a separate set of classes to represent the mobile code paradigm. These classes allow the programmer to instantiate active objects that import components to execute, then aggregate and carry the results of the executions of these components as they migrate throughout the network.

Through its messages, StratOSphere allows one to use different mobility models to write distributed programs. Each message must be statically created by the programmer, so component mobility in StratOSphere is explicit, and component migration policy and

an application's core logic are not separate. StratOSphere programmers may be able to achieve some runtime flexibility in the expression of their migration policies by using the polymorphism of the message class hierarchy to instantiate a message subclass appropriate for a given network configuration, such as instantiating a COD message upon one pass through a code block and a CS message upon another, while using a message abstract base class as a handle, but the authors do not address this possibility.

In StratOSphere, a computation may require more than one component. This is crucial to the definition of RCE above, which makes sense only if the computation requires components from different locations. Programmers using StratOSphere map the components to computations by fetching these components, by name, from a repository into their message. The StratOSphere literature does not address how the programmer specifies the order of execution of the components required by a computation. As StratOSphere's defines them, the reply to a `MobileObjectMessage` (COD) is not necessarily dispatched, while the result of dispatching an `RemoteEvaluationMessage` (REV) is not necessarily returned to the invoker/sender. Presumably, the programmer must dispatch the resulting message in the case of COD and manually return the results of an REV execution. The fact that dispatch and a return value is inherent to the CS message, but not the COD and REV messages, is a nonorthogonal property of StratOSphere's programming model.

MAGE has a simpler notion of computation. MAGE binds mobility attributes to a component to control where it executes, and thus which computation paradigm, COD or REV, governs invocations of that component. To emulate StratOSphere's RCE paradigm, a MAGE programmer would first send a component that invokes the other components in the RCE set. This component would bind a COD attribute to each of its references to the other components. As it invokes operations on the other components, MAGE would then lazily collocate them.

Step	Action	Description
0	`a.srStart`	starts the sender-receiver on node A
1	`a.controllerStart`	starts the controller on node A
2	`b.srStart`	starts the sender-receiver on node B
3	`b.controllerStart`	starts the controller on node B
4	`a.sendRequest`	A sends a request for the code required
5	`b.acceptRequest`	B accepts the request
6	`b.sendLMU`	B packs and tries to send the LMU
7	`b.examine`	B inspects the target node (A) to see whether it is trusted
8	`b.trusted`	B finds that A is trusted
9	`b.serialise`	B tries to serialise the LMU
10	`b.serSuccess`	B successfully serialises the LMU
11	`a.receiveLMU`	B sends the LMU / A receives it
12	`a.deserialise`	A deserialises the LMU
13	`a.deserialised`	the LMU is deserialised and checked for conflicts
14	`a.conflict`	a conflict is detected
15	`a.conflictResolved`	the conflict is resolved
16	`a.deserSuccess`	deserialisation process is successfully completed
17	`a.inspect`	LMU is inspected for security
18	`a.accepted`	it is accepted into the system
19	`a.deployLMU`	LMU is passed on to the application for deployment
20	`a.lmuAccept`	the application fully accepts it
21	`a.deployed`	LMU is successfully deployed on A

Table 8.1: COD in SATIN

8.4 SATIN

SATIN is a meta-model for applications hosted on mobile devices. A SATIN-compliant middleware must support dynamic reconfiguration of both behavior and layout. SATIN defines logical mobility, as distinct from the physical mobility of a cellphone, as "the migration of a partial or complete application or process from one host to another" [123, §1]. SATIN's unit of mobility is a logical mobility unit (LMU), which contains an arbitrary number of logical mobility entities (LME). In terms of MAGE, an LMU is a component, while a set of LME is analogous to the heap closure of the fields of a component. SATIN associates a set of attributes called properties with each LMU. These properties list the hardware and software dependencies of the LMU. For instance, they might specify that the LMU can only run a post Java 5 JVM.

Table 8.1, from [122, §1.3.3], shows how SATIN maps COD onto a sequence of actions: A and B are nodes, represented by `a` and `b` in the code. It is unclear whether Table 8.1 depicts model actions in the language of SATIN's meta-model or a sequence of statements in a SATIN-compliant middleware implementation. Even if Table 8.1 depicts model actions not code, its naïve realization as a programming model would be very low-level, even more so than classic agent, and with the attendant lack of isolating the mobility concern.

8.5 Columba

Columba discusses mobility in terms of binding [10]. Columba associates a *shadow* proxy each mobile device. A shadow proxy is mobile code, implemented using the Java-based Secure and Open Mobile Agent platform. A binder manager manages all of a shadow proxy's external references (bindings) and can dynamically and transparently adjusts those bindings, under the control of the policy manager. XML metadata describe each resource, such as whether the resource can move, its dependencies, and what operations it exports. Policies, written in the declarative Ponder language, use the metadata to determine how to change bindings. Columba allows administrators to add new or change existing policies at runtime. Columba realizes COD when an event triggers the binder manager to execute a "co-locality policy" which collocates two resources, which may be shadow proxies, then rewrites their previously remote bindings for each other with local bindings.

Columba offers two interesting features that MAGE does not: it 1) allows programmers, via its use of XML metadata, to write resource aware policies with less effort than MAGE's current implementation requires; and 2) supports injecting new policies into a running system. For the latter, MAGE assumes the set of policies (attributes) that a programmer can choose from is statically fixed. In contrast, MAGE 1) reports and analyzes its overhead, while Columba does not; 2) supports the composition of policies; and 3) uses a single language, Java, to express both application logic and migration policies. The fact that Columba separates the application logic and policy specification into two languages, Java and Ponder, hinders debugging (how can a programmer step through a shadow proxy when the binder

manager can rewrite its bindings?) and increases the cognitive scope of the programmer's task when using Columba.

8.6 Aspect Oriented Programming

Modularization as a means of managing complexity is one of software engineering's greatest triumphs. Aspect-oriented programming (AOP) [49] is an attempt to modularize cross-cutting concerns, those concerns that cannot be easily modularized because they overarch or frame different concerns. Logging and mobility, our particular concern, are two examples. AOP calls these cross-cutting concerns aspects and defines *joinpoints* as those points in the execution of a program where different concerns can be woven together. *Pointcuts* are named subsets of joinpoints; an example pointcut is the set of all method invocations in a program. Aspect weaving is usually static, but can be dynamic [78].

In AOP terms, MAGE dynamically weaves the mobility aspect into a distributed program at the method call pointcut and applies its attributes like before advice. AOP treats aspects as self-contained and focuses on their composition with the target code and with each other. Since the unit of composition in AOP is an aspect, an AOP programmer would be forced to represent each different migration policy as a separate aspect [103]. MAGE uses a single language, Java, for both the application logic and the mobility concern in the implementation of its attributes. The binding of attributes directly appears in the application logic, and is not expressed indirectly as a pointcut. Relative to AOP, this fact reduces the cognitive scope of MAGE, easing debugging and maintenance. MAGE focuses solely on the mobility concern. Freed of the need to support very general cross-cutting concern, MAGE is able to restrict its attributes and support their composition.

D^2AL [9] is a AOP model that focuses on the component mobility aspect of distributed programming. D^2AL provides programmers with a declarative language that specifies which components should be co-located and whether that co-location should be realized by movement or replication. D^2AL's proposed weaver works at compile time so D^2AL does not express runtime component migration policies; rather, it focuses on static program layout.

RoleEP [103] encapsulates component mobility in roles, and thereby addresses AOP's problem of how to separate migration policies without assigning them to different aspects. These roles dynamically bind to a component and extend that component with new methods that govern both how their migration policy and how other components can interact with the component. In RoleEP, a migration policy must be expressed separately from a role as a series of calls to the migration methods provided by that role. Mobility attributes in MAGE do not extend the interface of components to which they are bound; rather, they implicitly encapsulate a migration policy that the MAGE RTS applies upon each method invocation.

8.7 P2PMobileAgents

Like Columba, P2PMobileAgents [38] extends the explicit migration of traditional mobile agent programming models with metadata about the resources needed and provided by the agents and execution environment in the system and a declarative domain specific language for migration policies that selects an execution environment, given an agent's resource needs and the current network configuration. P2PMobileAgents uses XML to implement both of these extensions. It uses broadcasts in a P2P overlay network of execution environments to find agents and resources.

The authors do not address how to bind migration policies to agents, nor whether or not these policies can be composed. However, an example they give implies that, prior to an explicit migrate call, P2PMobileAgents executes the migration policy as a P2P query broadcast to the overlay network. Since queries must be built statically, it appears that, unlike MAGE, P2PMobileAgents' programming model is static, in the sense that it does not allow the rebinding of policies to components at runtime.

8.8 Summary

Table 8.2 lists whether and how the programming models discussed in this chapter isolate the mobility concern from the application logic. The difficulty associated with attempting

Programming Model	Separation of Concerns
Classic Agent	no
Columba	scripts
FarGo	scripts
MAGE	attributes
P2PMobileAgents	partial: target selection
RoleEP	roles
SATIN	no
StratoSphere	no

Table 8.2: Mobile Code Programming Models: Separation of Concerns

Programming Model	Binding	Trigger
Columba	resources	events
FarGo	*Complet*	events
MAGE	stub, component	invocation
P2PMobileAgents	n/a	n/a
RoleEP	objects	invocation

Table 8.3: Implicit Mobility Control

Programming Model	First Class	Resource-Aware	Language
Columba	no	application dependent	Java
FarGo	yes	yes	Java, Event-Action scripts
MAGE	yes	yes	Java
P2PMobileAgents	no	yes	declarative XML-based DSL
RoleEP	yes	application dependent	Java

Table 8.4: Migration Policy

to achieve this isolation strongly differentiates those that do not from MAGE, so we drop them from the tables that follow.

Although the remaining programming models all support explicit mobility through a `move` command, separating the mobility concern implies that a component's migration policy is not always manual. Table 8.3 summarizes to what the remaining programming models bind a migration policy and what triggers its application.

Table 8.4 characterizes the properties of the migration policies in each programming

model. By first class, we mean an entity that can be passed to method calls and modified within the language of implementation [95]. By resource aware, we mean a distributed system that tracks the resource needs of components and the availability of resources in the system [85].

Chapter 9

Conclusion

Don't look back. Something might be gaining on you.

Satchel Paige, 1906–1982

MAGE's *raison d'être* is that computation and resources must be dynamically co-located as resources appear and disappear and move around on a network. To realize this ambition, MAGE defines a programming model whose bedrock is the mobility attribute abstraction. This model defines expressions over mobile object, proxy, and its mobility attribute types.

Objects move when the application of which they are a part decides to move either the computation or the data that they represent from one namespace to another, usually for performance and efficiency reasons. In MAGE, an application makes its distribution wishes known via mobility attributes. Since, as we have shown, mobility attributes can encompass any distributed programming model and dynamically bind to program components, they allow the programmer who uses them to build flexible and adaptable distributed programs well-suited to today's dynamic and increasingly huge networks.

The principal contribution of this thesis is the MAGE programming model and its realization. A programmer would choose MAGE for projects that require dynamic layout adaptation because MAGE (1) separates the migration concern into attributes; (2) facilitates policy reuse via attribute composition; and (3) offers powerful, flexible, and elegant control

over object and invocation placement.

We believe that MAGE will provide the community a valuable framework for thinking about computation mobility. MAGE provides a high level of abstraction that unites the distributed programming models currently in widespread use, separates core application code from computation placement policies, and facilities the construction of complex migration policies using simple, well-understood policies as building blocks.

9.1 Future Work

The work begun in this dissertation can continue in a number of directions.

Implementation: Usually, the performance gain of collocation motivates mobility. The current implementation of MAGE undermines this motivation because it uses RPC, albeit over the TCP/IP loopback stack, even when two mobile objects are collocated, as discussed in Section 6.1. Of possible implementation improvements, the conversion of invocations made by collocated mobile objects from RPC to local procedure calls is an important next step.

Applications: MAGE is a programming model whose purpose is to ease and facilitate a programmer's control over a distributed application's layout. Therefore, useful and interesting applications are the measure of MAGE. A load-balancing application could test server-side component mobility attributes. Similarly, an application that demonstrates the utility of attribute operators should be formulated and built. Finally, MAGE can move components to and from a compute farm in response to the cost of CPU cycles. For a long-running scientific application, such a migration policy could reduce time-to-completion without exceeding a budget.

Forwarding Pointers: The comparison of a traditional, centralized directory and forwarding pointers in Chapter 5 is analytic. I intend to implement the optimized algorithm and investigate its utility in mobile device applications, such as mobile TCP/IP. I am also interested in investigating the applicability of forwarding pointers to P2P.

Peripatetic Dining Philosophers (PDP): When I formulated PDP, my ambition was to formulate a problem that researchers could use to evaluate the performance and expressivity of languages that support mobility. I intend to port PDP to a representative set of mobile-code languages and use it to compare and contrast them.

References

[1] Miki Abu and Israel Ben-Shaul. A multi-threaded model for distributed mobile objects and its realization in FarGo. In *Proceedings of the 21th International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[2] Bruno Achauer. The DOWL distributed object-oriented language. *Communications of the ACM*, 36(9):48–55, September 1993.

[3] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 186–201, New York, NY, USA, December 1999. ACM.

[4] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[5] Sara Alouf, Fabrice Huet, and Philippe Nain. Forwarders vs. centralized server: an evaluation of two approaches for locating mobile agents. *Performance Evaluation*, 49(1-4):299–319, 2002.

[6] K. Arnold and J. Gosling. *The Java Programming Language Third Edition.* Addison Wesley, 2000.

[7] Y. Artsy and R. Finkel. Designing a process migration facility: the Charlotte experience. *IEEE Computer*, pages 47–56, September 1989.

[8] Earl Barr, Raju Pandey, and Michael Haungs. MAGE: A distributed programming model. In *Proceedings of the 21th International Conference on Distributed Computing Systems (ICDCS)*, April 2001.

[9] U. Becker. D^2AL: A design-based aspect language for distribution control. In *Position Paper at the ECOOP'98 Workshop on Aspect-Oriented Programming*, 1998.

[10] Paolo Bellavista, Antonio Corradi, Rebecca Montanari, and Cesare Stefanelli. Dynamic binding in mobile applications: A middleware approach. *IEEE Internet Computing*, 7(2):34–42, 2003.

[11] T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen, and A. Secret. The world-wide web. *Communications of the ACM*, 37(8):76–82, August 1994.

[12] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[13] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, February 1984. 2(1).

[14] Andrew P. Black and Y. Artsy. Implementing location independent invocation. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):107–119, 1990.

[15] Barry W. Boehm, Ellis Horowitz, Ray Madachy, Donald Reifer, Bradford K. Clark, Bert Steece, Winsor A. Brown, Sunita Chulani, and Chris Abts. *Software Cost Estimation with Cocomo II*. Prentice Hall PTR, Upper Saddle River, NJ, USA, January 2000.

[16] A. Carzaniga, P. Pietro, and G. Vigna. Designing distributed applications with mobile code paradigms. In *International Conference on Software Engineering*, pages 22–32, Boston, MA, May 1997.

[17] K.M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 1984.

[18] Chin-Chen Chang and Iuon-Chang Lin. The strategy of reducing the location update traffic using forwarding pointers in virtual layer architecture. *Computer Standards and Interfaces*, 25(5):501–513, 2003.

[19] Seon G. Chang and Chae Y. Lee. A selective pointer forwarding strategy for location tracking in Personal Communication Systems. In *Proceedings of INFORMS*, pages 344–351, 2000.

[20] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets, 2001.

[21] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? Technical report, IBM T.J. Watson Research Center, 1995.

[22] Michael J. Demmer and Maurice Herlihy. The Arrow distributed directory protocol. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, pages 119–133, London, UK, 1998. Springer-Verlag.

[23] P. J. Denning. Thrashing: Its causes and prevention. In *Proceedings AFIPS Joint Computer Conference*, volume 33, pages 915–922, 1968.

[24] Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

[25] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, June 1971.

[26] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software, Practice and Experience*, 21(8):757–785, 1991.

[27] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[28] Brian Ensink and Vikram Adve. Coordinating adaptations in distributed systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pages 446–455, 2004.

[29] Frederic Fitch. *Symbolic Logic. An Introduction.* Roland Press, 1952.

[30] Robert Joseph Fowler. The complexity of using forwarding addresses for decentralized object finding. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 108–120, New York, NY, USA, 1986. ACM.

[31] Munehiro Fukuda, Lubomir F. Bic, Michael B. Dillencourt, and Fehmina Merchant. Messengers: Distributed programming using mobile agents. *Journal of Integrated Design and Process Science*, 5(4):95–112, 2001.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. ISBN 0-201-63361-2.

[33] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[34] Gnutella. http://www.gnutelliums.com/.

[35] Richard Goering. Keynote: How multicore will reshape computing. *EETimes*, March 2007.

[36] Robert S. Gray, George Cybenko, David Kotz, Ronald A. Peterson, and Daniela Rus. D'Agents: Applications and performance of a mobile-agent system. *Software, Practice and Experience*, 32(6):543–573, May 2002.

[37] Sabine Habert and Laurence Mosseri. COOL: kernel support for object-oriented environments. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP)*, pages 269–275, New York, NY, USA, 1990. ACM.

[38] Klaus Haller and Heiko Schuldt. Using predicates for specifying targets of migration and messages in a peer-to-peer mobile agent environment. In *Mobile Agents 2001*, pages 152–168, 2001.

[39] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *Proceedings of the ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[40] Miguel Helft. For start-ups, web success on the cheap. *The New York Times*, November 8th 2006.

[41] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[42] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. Dynamic layout of distributed applications in FarGo. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE)*, pages 163–173, 1999.

[43] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit. System support for dynamic layout of distributed applications. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, 1999.

[44] Ophir Holder and Hovav Gazit. FarGo programming guide. Technical Report EE Pub 1194, Technion — Israel Institute of Technology, 1999.

[45] An-Cheng Huang and Peter Steenkiste. Building self-configuring services using service-specific knowledge. In *IEEE HPDC'04*, pages 45–54, 2004.

[46] Galen C. Hunt and Michael L. Scott. The Coign automatic distributed partitioning system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, pages 187–200, Berkeley, CA, USA, 1999. USENIX Association.

[47] Ravi Jain and Yi-Bing Lin. An auxiliary user location strategy employing forwarding pointers to reduce network impacts of PCS. *Wireless Networks*, 1(2):197–210, 1995.

[48] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[49] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, 2001.

[50] J. Kiniry and D. Zimmerman. A hands-on look at Java mobile agents. *IEEE Internet*, pages 21–30, July–August 1997.

[51] Donald E. Knuth. Structured programming with go to statements. *ACM Computing Surveys*, 6(4):261–301, 1974.

[52] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. Agent TCL: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July–August 1997.

[53] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering (TSE)*, 16:1293–1306, 1990.

[54] P. Krishna. *Performance Issues in Mobile Wireless Networks*. PhD thesis, Texas A&M University, August 1996. Chair-Dhiraj K. Pradhan and Chair-Nitin H. Vaidya.

[55] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementation of Java remote method invocation. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[56] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commununications of the ACM*, 21(7):558–565, 1978.

[57] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson Education, Inc, 2002.

[58] D.B. Lange, M. Oshima, G. Karjoth, and K. Kosaka. Aglets: Programming mobile agents in Java. In *Proceedings of the Worldwide Computing and Its Applications*, pages 253–266, 1997.

[59] Yi-Bing Lin and Wen-Nung Tsai. Location tracking with distributed HLR's [*sic*] and pointer forwarding. *IEEE Transactions on Vehicular Technology*, 47(1):58–64, February 1998.

[60] Barbara L. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *Proceedings of ACM Sigplan 1988 Conference on Programming Languages, Design and Implementation (PLDI)*, 23(7):260–267, June 1988.

[61] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 104–111, June 1988.

[62] Steve Lohr. A software maker goes up against Microsoft. *The New York Times*, February 24th 2007.

[63] W. Lugmayr. *Gypsy: A component-oriented mobile agent system.* PhD thesis, Technical University of Vienna, October 1999.

[64] Scott Malabarba, Raju Pandey, Jeffrey Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, 2000.

[65] John Markoff. Intel prototype may herald a new age of processing. *The New York Times*, February 12th 2007.

[66] Sun Microsystems. Java Remote Method Interface (RMI). java.sun.com/products/jdk/rmi/index.

[67] Sun Microsystems. Object Serialization Specification. Available at http://java.sun.com/products/jdk/1.2/docs/guide/serialization/spec/serialTOC.doc.html.

[68] Sun Microsystems. Java Management Extensions (JMX). Available at http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/, September 2007.

[69] Sun Microsystems. Trail: RMI. Available at http://java.sun.com/docs/books/tutorial/rmi/index.html, July 25 2007.

[70] Sun Microsystems. `Future<V>`. Available at http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Future.html, February 2008.

[71] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration survey. *ACM Computing Surveys*, 2000.

[72] Luc Moreau. Distributed directory service and message router for mobile agents. *Science of Computer Programming*, 39(2–3):249–272, 2001.

[73] Luc Moreau. A fault-tolerant directory service for mobile agents based on forwarding pointers. In *The 17th ACM Symposium on Applied Computing (SAC) — Track on Agents, Interactions, Mobility and Systems*, pages 93–100, Madrid, Spain, March 2002.

[74] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, 1999.

[75] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS)*, pages 22–30, 1982.

[76] Athanasios Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw Hill, 3rd edition, 1991.

[77] David Patterson, Arvind, Krste Asanovic, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Mark Oskin, Jan Rabaey, and John Wawrzynek. RAMP: Research accelerator for multiple processors. In *Proceedings of the 18th Hot Chips Symposium*, August 2006.

[78] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gérard Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of Reflection 2001*, pages 1–24, 2001.

[79] H. Peine and T. Stoplmann. The architecture of the Ara platform for mobile agents. In *Proceedings of the First International Workshop on Mobile Agents (MA)*, 1997.

[80] Gian Pietro Picco. μCode: A lightweight and flexible mobile code toolkit. In K. Rothermel and F. Hohl, editors, *Mobile Agents, Proceedings of the 2nd International Workshop on Mobile Agents (MA)*, volume 1477, pages 160–171, Stuttgart (German), September 1998. Springer.

[81] José M. Piquer. Indirect distributed garbage collection: handling object migration. *ACM Transactions on Programming Languages and Systems*, 18(5):615–647, 1996.

[82] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[83] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. *SIGOPS Operating System Review*, 17(5):110–119, 1983.

[84] Dongyu Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 367–378, New York, NY, USA, 2004. ACM.

[85] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware mobile programs. In *Proceedings of the 1997 USENIX Annual Technical Conference*, 1997.

[86] Todd Rowland and Eric W. Weisstein. Characteristic function. http://mathworld.wolfram.com/CharacteristicFunction.html, August 2007. From MathWorld–A Wolfram Web Resource.

[87] V. Ryan, S. Seligman, and R. Lee. Schema for representing Java™ objects in an LDAP directory. RFC 2713 (Informational), October 1999.

[88] Jürgen Schwille. Use and abuse of exceptions - 12 guidelines for proper exception handling. In *Ada-Europe '93: Proceedings of the 12th Ada-Europe International Conference*, pages 141–152, London, UK, 1993. Springer-Verlag.

[89] George Semeczko and Stanley Y. W. Su. Supporting object migration in distributed systems. In *Proceedings of the 5th International Conference on Database Systems for Advanced Applications*, pages 59–66, Melbourne, Australia, April 1997.

[90] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 135–146, New York, NY, USA, 1992. ACM.

[91] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Technical Report 1799, INRIA, 1992.

[92] J. Siegel. *CORBA: Fundamentals and Programming.* Wiley, 1996.

[93] J.W. Stamos and D.K. Gifford. Remote evaluation. In *ACM Transactions on Programming Languages and Systems*, volume 12, pages 537–565, October 1990.

[94] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, San Diego, California, August 2001.

[95] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[96] M. Strasser, J. Baumann, and F. Hohl. Mole: A Java based mobile agent system. In *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, 1997.

[97] Andrew Tanenbaum. *Computer Networks*. Prentice Hall, 3rd edition, 1996.

[98] The Coq Development Team. The Coq proof assistant reference manual. Technical report, INRIA, 2009.

[99] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[100] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, Malaga, June 2002.

[101] Anand Tripathi, Neeran Karnik, Manish Vora, Tanvir Ahmed, and Ram Singh. Mobile agent programming in Ajanta. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, May 1999.

[102] Anand R. Tripathi, Neeran M. Karnik, Tanvir Ahmed, Ram D. Singh, Arvind Prakash, Vineet Kakani, Manish K. Vora, and Mukta Pathak. Design of the Ajanta system for mobile agent programming. *Journal of Systems and Software*, 62(2):123–140, May 2002.

[103] Naoyasu Ubayashi and Tetsuo Tamai. Separation of concerns in mobile agent applications. In *Proceedings of the 3rd International Conference Reflection 2001, LNCS 2192*, pages 89–109. Springer, 2001.

[104] Peter Wayner. Cloud versus cloud: A guided tour of Amazon, Google, AppNexus, and GoGrid. *InfoWorld*, July 21 2008.

[105] Eric W. Weisstein. Machin's formula. http://mathworld.wolfram.com/MachinsFormula.html, 2007. From MathWorld — A Wolfram Web Resource.

[106] David A. Wheeler. Linux kernel 2.6: It's worth more! http://groklaw.net, October 2004.

[107] K. Whisnant, Z. T. Kalbarczyk, and R. K. Iyer. A system model for dynamically reconfigurable software. *IBM System Journal*, 42(1):45–59, 2003.

[108] Jim E. White. Telescript technology: The foundation for the electronic marketplace. General Magic Inc. White Paper, 1994. http://www.magic.com.

[109] Jim E. White. Mobile agents. General Magic Inc. White Paper, 1996. http://www.magic.com.

[110] Wikipedia. Adobe Flash. http://en.wikipedia.org/wiki/Adobe_Flash, 2007.

[111] Wikipedia. Indicator function. http://en.wikipedia.org/wiki/Indicator_function, August 2007.

[112] Wikipedia. Mozilla JavaScript. http://en.wikipedia.org/wiki/JavaScript, 2007.

[113] Wikipedia. Operational semantics. http://en.wikipedia.org/wiki/Operational_semantics, August 2007.

[114] Wikipedia. Sun grid. http://en.wikipedia.org/wiki/Sun_Grid, July 2007.

[115] Wikipedia. Web 2.0. http://en.wikipedia.org/wiki/Web_2, August 2007.

[116] Wikipedia. Directed acyclic graph. http://en.wikipedia.org/wiki/Directed_acyclic_graph, November 2008.

[117] Wikipedia. Livelock. http://en.wikipedia.org/wiki/Deadlock#Livelock, September 2008.

[118] Daniel Wu, Divyakant Agrawal, and Amr El Abbadi. StratOSphere: Mobile processing of distributed objects in Java. In *Proceedings of the 4th Annual ACM/IEEE Interna-*

tional Conference on Mobile Computing and Networking (MobiCom), pages 121–132, Dallas, TX, October 1998.

[119] Daniel Wu, Divyakant Agrawal, and Amr El Abbadi. Mobility and extensibility in the StratOSphere framework. *Distributed and Parallel Databases*, 7(3):289–317, 1999.

[120] Daniel Wu, Divyakant Agrawal, and Amr El Abbadi. StratOSphere: Unification of code, data, location, scope, and mobility. In *Proceedings of International Symposium on Distributed Objects and Applications (DOA)*, Edinburgh, Scotland, September 1999.

[121] YourKit, LLC. www.yourkit.com, January 2009.

[122] Stefanos Zachariadis, Manish Lad, Cecilia Mascolo, and Wolfgang Emmerich. Building adaptable mobile middleware services using logical mobility techniques. In *Contributions to Ubiquitous Computing*, pages 3–26. Springer-Verlag: Heidelberg, Germany, 2007.

[123] Stefanos Zachariadis, Cecilia Mascolo, and Wolfgang Emmerich. The SATIN component system — a metamodel for engineering adaptable mobile systems. *IEEE Transactions on Software Engineering (TSE)*, 32(11):910–927, November 2006.