MAGE: A Distributed Programming Model*

Earl Barr Raju Pandey Michael Haungs
Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
{barr, pandey, haungs}@cs.ucdavis.edu

Abstract

Writing distributed programs is difficult. To ease this task, we introduce a new programming abstraction, which we call a mobility attribute*. Mobility attributes provide a syntax that describes the mobility semantics of program components. Programmers attach mobility attributes to program components to dynamically control the placement of these components within the network. Mobility attributes intercept component invocations and decide whether and where to move a component before the component executes. This allows the programmer to improve her program's runtime efficiency by colocating components and resources. We present MAGE, an object oriented distributed system, that supports mobility attributes and illustrates their utility.*

1 Introduction

Today, a wide variety of services and data exist dispersed on architectures that are heterogeneous and evolving. The Web [6] exemplifies this trend. Large scale scientific computation is another such service: it is moving from its traditional super computer environment to a distributed one, alured by the extensibility and cost savings that distributed systems offer. Indeed, new companies have formed that capitalize on this trend by renting out processor pools or farms [1].

The distributed systems that support these services must handle distributed, dynamic and moving processing and data resources: over time, a host whose CPU was pegged may become idle and one data source may be exhausted while another comes online. Since the network infrastructure on which these systems run is also dynamic, with systems joining and crashing, these systems must also be extensible. They must support host and resource discovery, incorporate new hardware and robustly cope with changing network conditions. To fully exploit this runtime environment, distributed programming models must provide mechanisms that allow programs to migrate their components, support load balancing, respond to network congestion and adapt to the appearance, disappearance and shifting of resources.

Recognizing this, researchers have proposed distributed programming models that support various forms of code and data migration. We seek to extend and unify their work with a new programming abstraction, called a *mobility attribute*, that represents the distribution aspects of program components as first class objects.

In this paper, we present a distributed programming model and its implementation, called Mobility Attributes Guide Execution (MAGE), based on mobility attributes. MAGE uses mobility of program components (classes, methods and objects) as a basis for managing the complexity of the underlying execution environment. MAGE attaches mobility attributes to components and thereby controls component migration. Thus, programs can create specific distribution patterns by binding specific mobility attributes with their components. Programs can also dynamically rebind mobility attributes to modify their distribution characteristics, as their runtime environment evolves. The MAGE runtime system transparently manages location of code and data, and arranges for the execution of specific program components.

The MAGE programming model differs from most existing distributed programming models. It

1. allows programmers to write distribution policies that attach to components and define the actions taken whenever their application invokes a component,

*This work is supported by NSF grant no. CCR-0082677 and by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Project Agency (DARPA), Rome Laboratory, or the U.S. Government.

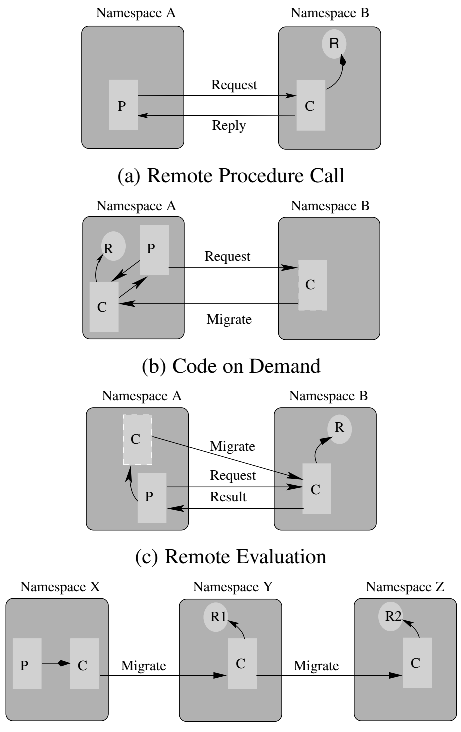
2. permits programmers to separate their application logic from the exigencies of network programming,

3. extends distributed programming models that make static assumptions about component placement to mobile components, and

4. encapsulates and expresses current distributed programming models, thereby unifying them as well as allowing the easy formulation of new ones.

The rest of the paper is organized as follows: In Section 2, we review and analyze existing distributed programming models, before introducing mobility attributes and the MAGE programming model in Section 3. We then describe our implementation of MAGE and some of the more interesting issues that arose during the implementation in Section 4. If applications constructed using mobility attributes incur too great a performance penalty, mobility attributes would have no practical utility. In Section 5, we measure their overhead and present an illustrative example of their use. In Section 6, we place MAGE into the context of its related work and conclude, in Section 7, with some remarks about MAGE's limitations and its future direction.

2 Distributed Programming Models



(a) Remote Procedure Call

(b) Code on Demand

(c) Remote Evaluation

(d) Mobile Agent

Figure 1. Distributed Programming Models: C is a distinguished component of a program, P represents the program's remaining components, and R is a resource. C requires resource R.

In this section, we review widely used distributed programming models and compare them. One can classify these models into four categories [9] (RPC, COD, REV, and MA), which we review below. Figure 1 accompanies the review, focusing on the mobility semantics that each model imposes on a program's invocation of its components. In this figure, a namespace is an execution environment that defines name to component bindings.

A *local procedure call* (LPC) occurs when a component invokes another component in the same namespace. LPC is, of course, as old as modular programming and usually assumed in distributed settings. We explicitly include LPC because programmers employ it in distributed systems wherever possible because of its inherent efficiency.

A *remote procedure call*[1] (RPC) [7] arises when a client invokes a remote component. This component must already reside on the computation target. If necessary, a stub that handles parameter marshalling is sent to the invoking namespace. Java's RMI [2] is an instance of the RPC model.

Code on demand (COD) denotes a local computation that requires a remote component, which is downloaded to the local namespace. Java applets [4] are a popular implementation of this model.

Remote evaluation (REV) [24] occurs when a client desires the remote execution of a local component. `P` moves component `C` to the desired namespace `B`, where the computation occurs, as shown in Figure 1(c).

Mobile Agent (MA) [10, 26] describes a component that can move itself, while it is executing, from one namespace to another.

Alone, each of the models discussed above has drawbacks. Each model tackles complexity by restricting its application to certain network configurations. RPC, for instance, requires static knowledge of its remote component's location and, as a result, forces the programmer to statically distribute application code. Both COD and REV provide mobility, but only of code whose location the programmer must statically know. Further, they both support extensibility, but only in one direction. COD moves code into clients and thereby extends their functionality, while REV extends servers. MA, since it moves computation state, is heavyweight.

To surmount these drawbacks, most distributed systems support more than one of these models, but even when they do, they permit only static binding of a model to a given in-

[1] [9] call this model Client-Server.

vocation. Dynamically combining these models would allow programmers to handle a wider range of network configurations.

3 Mobility Attributes

In this section, we define mobility attributes. We then reexamine the classical distributed programming models (see Section 3.3) and unify them with mobility attributes. As an example of the expressive power of mobility attributes, we use them to define a new distributed programming model. We then discuss what happens when the system state, by which we mean the application, network and MAGE system state, does not match the system state the mobility attribute expects. We conclude with an illustrative example.

3.1 Overview

Mobility attributes are first class objects that bind to program components. A mobility attribute intercepts invocation requests on the components to which it has been bound. For a given network configuration, mobility attributes describe where their component should execute. If necessary, the component moves before executing.

In our current implementation, mobility attributes define a bind method that moves the component and returns a stub to the programmer. So a programmer can define a migration policy based on load, by defining a mobility attribute with the following bind implementation:

```
public Remote bind() {
  if ( cloc.getLoad() > 100 ) {
    target = selectNewHost();
    cachedStub = send(target);
  }
  return cachedStub;
}
```

The programmer then instantiates this mobility attribute, `ma` and employs it by calling its `bind` method prior to invoking a method on its component. We currently rely on the programmer to manually enforce the binding semantics of a mobility attribute by calling the mobility attribute's bind method prior to invoking a method on the bound class (Please see Section 4 for more details.):

```
o = ma.bind();
o.f();
```

We believe that component invocation is a natural place to decide where the component should execute, since the application can apply its detailed knowledge of how best to use and acquire the resources it needs, given its state and the current state of the network.

	Current Location	Target	Moves Component
MA	remote	remote	yes
REV	local	remote	yes
RPC	remote	remote	no
CLE	not specified	not specified	no
COD	remote	local	yes
LPC	local	local	no

Table 1. Distributed Programming Models Parameterized.

3.2 Definition

All distributed programming models specify a network configuration and a target. Divorced from their assumptions about system state, each model essentially specifies a namespace. Additionally, each classical model implicitly specifies the component's current location and mobility. Our notion of mobility attributes arose from this insight: Like the programming models they unify, mobility attributes specify a current location, computation target and whether or not the bound component should move.

Consider Table 1. This table captures the salient features of the conventional distributed programming models mentioned in Section 2[2]. These features define the design space. The triple $< Location, Target, Moves >$, where Location, Target $\in$ {remote, local, not specified} and Moves $\in$ {yes, no}, uniquely specifies all distributed programming models discussed in this paper. For example, the triple $< remote, local, yes >$ concisely and uniquely defines COD. Thus, mobility attributes are instances of these triples.

3.3 Defining Programming Models

As defined above, mobility attributes also allow us to see relationships between the models that perhaps weren't obvious before. For instance, when a component's current location is the same as the target; that is, if there is no need to move the component because it's already at the target, REV becomes RPC. Mobility attributes allow us to make use of these observations by giving us the means to generalize distributed programming models. Indeed, mobility attributes make it easier to think of, define and experiment with new distributed programming models. Below, we illustrate this point by generalizing REV and introducing a new model.

In MAGE, we can define a mobility attribute, GREV, that generalizes REV and can be used in place of both REV and

[2]We ignore parameters to the component invocation here: we assume that the necessary parameters can be sent to the target prior to invocation using the traditional data marshalling mechanisms.

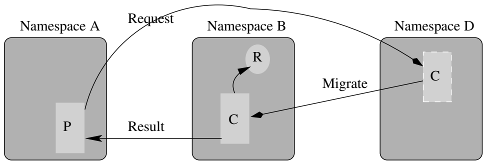


Figure 2. Generalized Remote Evaluation

COD: GREV moves its component to its target, regardless of whether the component was initially local or remote and whether the target is local or remote. While more expensive than either REV or COD, GREV applies to a wider array of component distributions than either REV or COD alone and is well suited to distributed systems in which components are constantly moving. Figure 2 illustrates the behavior of GREV. `P` requests component `C` move from its current namespace `D` to the computation target `B`, where the computation occurs. When the computation completes, `P` receives the result. To realize GREV, we define a mobility attribute that accepts any namespace as its component's initial location and target.

All the entries in Table 1 we have discussed so far specify their computation target. What if the program does not care where a component executes at some point in time? We introduce a new distributed programming paradigm model, *Current location evaluation* (CLE)[3], that answers this question. CLE does not specify a computation target; rather, CLE evaluates its component in the namespace in which the component currently resides. Thus, CLE does not express mobility, but at the same time only makes sense in the context of mobile components, which must be found. To realize CLE, we simply define a mobility attribute whose target is the set of all namespaces on the network.

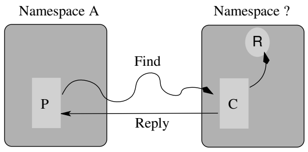


Figure 3. Current Location Evaluation

To elucidate CLE's utility, consider a printer management program consisting of clients, print servers and a job controller. In the unlikely event that users did not care which printer they used, clients could fruitfully use CLE to invoke a print server component while the job controller moved the print server components around the network in response to printer availability. In Figure 3, `P` finds `C` to make its invocation request. MAGE migrates computations, while Java's Jini[29] migrates code. Thus, CLE differs from Jini in that it can refer to the same component across invocations and namespaces. Jini refers to the same functionality or interface, but must destroy and create new objects when moving that functionality from one namespace to another.

We can also use MAGE to define mobility attributes that restrict the namespace on which a component can execute by restricting current location and target to subsets of the available hosts. Thus, mobility attributes not only unify the existing models, they are capable of expressing all models in the design space.

3.4 Mobility Coercion

A mobility attribute can specify component migration that does not make sense, as when applying COD to a component that is already local. These mismatches arise because of component mobility. Consider an invocation that applies a mobility attribute that defines traditional REV to a component that is already at the target. MAGE could either simply invoke that component or notify the application. To handle these mismatches, we propose *mobility coercion*. Whenever a mismatch occurs, MAGE attempts to coerce the computation into a distributed programming paradigm that matches the actual distribution of code and data. Table 2 describes programming model behavior for different scenarios.

3.5 Mobility Attribute Class Hierarchy

MAGE provides mobility attributes that implement the most commonly used distributed programming models as objects instantiated on the class hierarchy depicted in Figure 5. The root of this hierarchy is the following abstract class:

In our current implementation, the `bind` method interacts with the MAGE RTS (See subsection 4.1) to find its component, select a computation target and move the component to that target. The `find` method above is used to find objects shared by several threads. Since the object is shared, it may have been moved by another thread in between invocations by the current thread and must, therefore, be found before the current thread invokes it (See Subsection 4.4 for more details). If the object is private, `cloc` always accurately represents the bound object's current location in the network.

We must always cast `bind` invocations because Java does not currently support genericity. The `bind` method also defines the mobility attribute's behavior under mobility coercion. Mobility attributes differ mainly in their im-

[3] CLE is similar to the RCE model in StratOSphere [31]

	Component Location		
	Local	Remote	
		At Computation Target	Not At Computation Target
MA	Default Behavior	RPC	Default Behavior
REV	Default Behavior	RPC	Default Behavior
COD	LPC	n/a	Default Behavior
RPC	Exception thrown	Default Behavior	Exception thrown
CLE	Default Behavior	Default Behavior	Default Behavior

Table 2. Component Location and Programming Model Behavior

```
public class MobilityAttribute {
  Location target;
  Location cloc;
  String name;

  public MobilityAttribute(String t, String n) {
    target = t;
    name = n;
    cloc = find(name);
  }

  public Location find(String name) {...}
  public boolean isShared(String name) {...}

  public Remote bind(String n) {
    name = n;
    return bind();
  }

  public abstract Remote bind();
}
```

Figure 4. The Mobility Attribute Abstract Class

plementations of this `bind` method. For example, COD bind looks like

```
public Remote bind() {
  if (isShared(name)) {
    currentLocation = find(name);
  }
  return cloc.getObject(name);
}
```

There are two forms of migration in the MA paradigm — *weak* and *strong*. Strong migration moves a thread's stack along with heap state, while weak migration just moves heap state. Since the standard Java virtual machine does not provide access to execution state, MAGE uses weak migration. Thus, REV and MA differ under MAGE in that REV is single hop and synchronous, while MA is multi-hop and asynchronous.

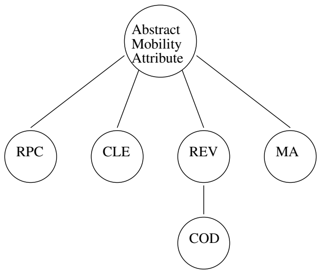


Figure 5. The Mobility Attribute Class Hierarchy

3.6 Example

Here we provide an example that illustrates how a program might use mobility attributes to dynamically adapt and react to the changing distribution of resources on a network.

Consider an oil company exploring for oil. This company has deployed sensors to gather geologic data that it will use to determine where to drill. These sensors are generating an enormous amount of data, which we would like to filter in place, at the sensor. We have an interface, `filter`, to an object, called `geoData`, which is an instance of a `GeoDataFilterImpl` that implements the `GeoDataFilter` interface. The object `geoData` knows how to gather and filter the data. We declare an REV mobility attribute and call its bind to instantiate `geoData` on its target, `sensor1`, as follows:

```
REV rev = new REV("GeoDataFilterImpl", "geoData",
                  "sensor1");
filter = (GeoDataFilter)rev.bind();
filter.filterData();
```

When `sensor1` is exhausted, we move `geoData` to `sensor2` with

```
MAgent magent = new MAgent("geoData", "sensor2");
filter = magent.bind();
filter.filterData();
```

Finally, we'd return the data to our research lab by binding a COD mobility attribute to the `geoData` object where we would process its results locally (Refer to Subsection 4.2 for a discussion of the semantics of binding COD to an object):

```
COD cod = new COD("geoData"); //target is local
filter = cod.bind();
filter.processData();
```

Because the MAGE RTS transparently handles component discovery for the programmer, it allows programmers to use mobility attributes that encode programming models that assume static distribution of code and data, with mobile components. In other words, programmers can reason about their applications using the simpler semantics of the static models, while still using mobile components. We see this in the example above where we can bring `geoData` back to the lab by applying COD without worrying about which sensor is currently hosting `geoData` when we invoke `processData`.

We could also simplify our example by defining our own mobility attribute, `CombinedMA`, which combines the above steps into one, fine-grained migration policy. This mobility attribute would contain the three mobility attribute declared above and its `bind` method would look something like

```
public Remote bind() {
  target = selectTarget(status);
  if (target.equals("sensor1))
    return rev.bind(name);
  else if (target.equals("researchLab"))
    return cod.bind(name);
  else
    return magent.bind(name);
}
```

With this mobility attribute, the above code could be rewritten as

```
CombinedMA combinedMA = new
  CombinedMA("GeoDataFilterImpl", "geoData");
while (iterator.moreSensors()) {
  filter = (GeoDataFilter)combinedMA.bind();
  filter.filterData();
}
filter = (GeoDataFilter)combined.bind();
filter.processData();
```

As we can see, this fragment is more compact and general than the code it replaces. It seamlessly handles the addition of new sensors. It loops through a list of sensors and applies a single mobility attribute that controls where `geoData` executes across all method invocations on `geoData`. This code snippet illustrates how mobility attributes encapsulate distribution logic in their bind method. Since programmers can define their own mobility attributes, such as `combinedMA` above, they can use mobility attributes to control the placement of their components, while keeping their application code clean, spare and focused on its problem domain. Thus, this example illustrates how mobility attributes give programmers the benefit of location transparency without loss of control over the placement of program components.

4 Implementation

In this section, we discuss MAGE RTS library and how we implemented mobility attributes. We then turn to the REV protocol as an illustrative example of how MAGE moves an object and conclude with the MAGE locking mechanism.

4.1 The MAGE RTS

Cooperating Java virtual machines (JVMs) comprise MAGE; these JVMs layer a homogeneous and consistent programming environment over the underlying heterogenous network hardware. The MAGE services employ Java RMI to communicate across the network. MAGE addresses security issues by employing Java's full panoply of security measures.

The MAGE RTS overlays the JVM with a collection of objects that the user's application instantiates at startup. These objects include a Mage registry and objects that implement the MAGE system's remote, or `MageExternalServer`, and home, or `MageServer` interfaces. The system is depicted in Figure 6. In the figure, the hexagons denote mobility attributes while the circles denote objects. The letters signify the names of the objects, shared by both the objects and the mobility attributes bound to them.

The `MageServerImpl` class implements `MageServer` and communicates with local mobility attributes. The `MageExternalServerImpl` class implements `MageExternalServer`. This class defines the methods used to send and receive object and classes, as well as forward registry requests. On the behalf of mobility attributes, these classes query the registry, lock objects to their current namespace and cooperate to move objects and classes.

The MAGE Registry wraps the RMI registry and tracks object locations. It also caches classes. For mobile objects, the registry maintains a list of all the objects that have ever been moved into a namespace in the registry's JVM and their last known location. To find an object, the registry simply follows the chain of forwarding addresses until its reaches the MAGE server currently hosting the component. As the result returns, each server updates its forwarding address, thus collapsing the path. Thus, the MAGE Registry

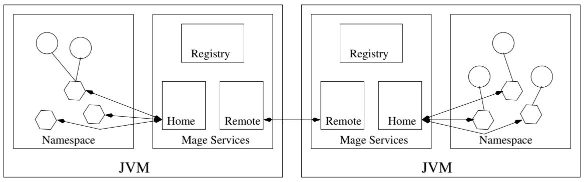


Figure 6. The MAGE System

defines a global, system-wide namespace for both mobile objects and classes.

4.2 Mobility Attribute Implementation

In Java, objects cannot exist without classes, but classes can exist without objects. Thus, a class and an object form a pair, whose object can be null. MAGE maps its notion of component to this pair. In other words, MAGE binds mobility attributes to both classes and objects. Binding components to a class becomes a convenient way to instantiate and move objects with one construct.

The mapping of components to both objects and classes alters the implementation semantics of REV and COD. Traditionally, REV and COD move a class to their target where they instantiate an object. Thus, REV and COD are object factories as traditionally defined. MAGE supports this definition, but since MAGE allows mobility attributes to bind to objects as well, MAGE allows the application of REV and COD to objects. This allows another definition of REV and COD where they move an existing object. A further definition that MAGE also supports is REV and COD as a single use factories. Under this definition, when REV and COD are applied to a class they behave traditionally, but then bind to the object that they instantiated. On subsequent invocations, this form of REV and COD would then move the object they first instantiated instead of instantiating new objects: in other words, they behave according to the object definition above.

Since REV coerces to RPC and objects are mobile in MAGE (Section 3.4), it would seem that an RPC mobility attribute is not necessary. We provided one anyway so that a programmer could use it to denote an immobile object. MAGE RPC throws an exception if it does not find its object on its target.

MAGE currently clones classes, leaving behind a copy of each object's class that visited a particular node. This means that MAGE implicitly defines mobile classes globally. Caching class definitions in this way is an optimization that can speed up object migration. Obviously, this scheme is not well-suited for classes with static fields, nor does it scale well. Handling classes with static fields would require extending MAGE to provide coherency for class data.

To keep our presentation focused on mobility attributes, MAGE's essential contribution, we choose to implement a simple object model. In MAGE, objects exist in only one namespace at a time. MAGE does not partition their state across namespaces, nor does MAGE clone them. MAGE objects can be public or private. If they are public, they can be accessed by more than one thread of execution and require locking as discussed in Subsection 4.4. MAGE uses RMI to support remote references to these objects using handles, or Java interfaces, that point to stubs.

Since MAGE is built on top of RMI, mobility attributes boil down to RMI calls. Their bind method is, in essence, a complex wrapper for RMI's `Naming.lookup` method. It uses RMI calls to find the object to which it is bound, move that object and return a stub. Thus, the RPC mobility attribute is a very thin wrapper of a standard RMI call, since it simply returns a stub, while REV uses RMI to perform all three operations.

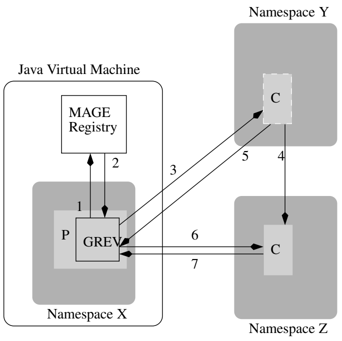


Figure 7. The GREV Protocol

4.3 GREV Protocol

The RMI calls employed by a mobility attribute's bind method define its messaging protocol. Figure 7 depicts the protocol used by an GREV mobility attribute to accomplish its task, when its object `C` is remote but not yet at its computation target. The mobility attribute, denoted `GREV`, finds `C` by consulting the local MAGE registry, at `1` and `2`. The figure elides any messages sent by the registry in the course of finding `C`. After `GREV` determines its computation target, it sends message `3` to the remote virtual machine to move `C` from namespace `Y` to `Z`. `Y`'s virtual machine sends `C` at `4`, then informs `REV` with the message `5`. `GREV` then invokes the operation on `C` by sending message `6` and receives its result in `7`.

Since mobility attributes can direct the MAGE RTS to send messages to find, move and invoke components, mobility attributes implicitly define protocols, just as the distributed programming models they encompass. These protocols must recover from message loss and account for contention over shared components. Thus, mobility attributes allow programmers to define their own invocation protocols.

4.4 Locking Shared Objects

In MAGE, two distinct, nearly simultaneous invocations can apply different mobility attributes to an object. These different mobility attributes may choose different target namespaces to which to move the object. Object movement, as Subsection 4.3 makes clear, is not atomic. If we allowed the two mobility attributes to interleave their movement operations, the result would be unpredictable: the object could be cloned or moved before an invocation completes.

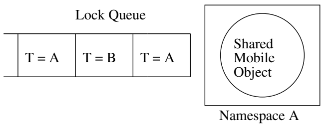


Figure 8. Mobile Object Locking

Thus, if `A.f` and `B.g` both invoke `C.g`, MAGE must ensure their mutual noninterference. To this end, MAGE employs locks, as shown in Figure 8. Each mobile object has a lock queue. Each lock request in the queue carries its mobility attribute's computation target, `T`. If the mobile object already resides in the namespace named by the lock request, MAGE returns a `stay` lock to the requesting mobility attribute, otherwise it returns a `move` lock[4]. Because object migration is so expensive, MAGE's current locking implementation unfairly favors invocations that `stay` lock their object.

The following fragment continues our oil exploration example begun in Subsection 3.6 and illustrates how MAGE brackets an invocation with locking. The `lock` method takes the name of the object and the mobility attribute's target, which it uses to determine whether to acquire a `stay` or `move` lock.

```
lock("geoData", cod.getTarget());
i = (GeoDataFilter) cod.bind();
x = i.f(a);
unlock("geoData");
```

[4] `Stay` and `move` locks are simply read and write locks under another guise. Also, MAGE locks layer Java synchronization mechanisms, if they are present. So access to a synchronized method of a mobile object would still be synchronized among those readers who shared that object's MAGE lock.

Distributed Programming Model	Single Invocation Time (ms)	Amortized (10) Invocation Time(ms)
Java's RMI	33	20
Mage's RMI	34	23
Traditional COD (TCOD)	66	22
Traditional REV (TREV)	130	82
MA	110	63

Table 3. MAGE Overhead Measurements

5 Experiments

For mobility attributes to be at all practical, they must not impose too much overhead upon their user. In this section, we report the overhead of using MAGE and discuss whether or not this overhead is prohibitive.

Our experimental testbed consists of two dual-processor 450 mhz pentium III machines connected via standard 10 Mb/s Ethernet. Each machine has 256mb of RAM and runs Linux 2.2.16. We use Sun's JDK 1.2.2.

We measure four popular distributed models implemented with mobility attributes in the MAGE distributed system. The four models are: RMI, traditional COD, traditional REV, and MA. These models are described in Section 2. To provide a frame of reference, we also measure the overhead of Java's RMI. For TCOD, the test object's class file (a minimal extension of UnicastRemote) is migrated to the local host, the local host instantiates a test object and invokes the appropriate method. This class has a single integer attribute, which it increments, so its marshalling overhead is minimal. Finally, the results are returned (local). For TREV, we do the reverse. The class file is local and migrated to the remote host where it is instantiated and invoked. The result is sent back to the local host. MA is similar to TREV except that the result stays at the remote host.

The measurements are contained in Table 3. We give single invocation times and amortized (the average of 10 invocations) in the second and third columns, respectively. The single invocation times show the one-time startup cost of priming the MAGE engine (warming the caches) while the amortized times give a more accurate representation that realistic MAGE applications will experience. Thus, we will now only discuss the amortized times.

We can see from Table 3 that the time reported for MAGE's implementation of the well-known distributed models are multiples of the time for Java's RMI. This is expected, as (1) MAGE is implemented on top of Java's RMI and (2) MAGE's implementation of TCOD, TREV, and MA involve multiple calls to Java's RMI. For example, MAGE's RMI is a thin wrapper for Java's RMI and therefore experiences only a slightly longer execution time.

Also, REV involves four Java RMI calls in our implementation of MAGE. Java's RMI is obviously the dominant cost in our MAGE implementation. MAGE would directly benefit from having a more optimized Java RMI implementation [20] and condensing the number of RMI calls in the MAGE implementation. This condensing can be acheived by better utilizing the `in` and `out` variables of a single Java RMI call. Being even more ambitious, we could bypass this overhead by implementing our own migration protocol directly with TCP/IP. This would allow us to directly and efficiently exploit the migration semantics of the various models without retrofitting them onto RMI.

6 Related Work

The idea of supporting program mobility is not new and has appeared in various forms in distributed operating system [5, 11, 19] and programming language [15, 14] research. Broadly, this research has explored systems that offer ever greater degrees of mobility, progressing from the date migration inherent to RPC [7] to explosion of interest in MA [26]. In this section, we survey both the earlier work and recent advances in program mobility.

6.1 Data and Code Migration

Historically, RPC-based systems have assumed static distribution of components and their definitions. Java's RMI [2], CORBA [23], and COM/DCOM [12] exemplify such RPC-based distributed system infrastructures. Recently, systems, such as Jini [29] and the Ninja project's Multispace [13], have augmented RPC with mechanisms for distributing code, using some form of REV. In these systems, applications can discover resources, push code for these resources to other hosts, and perform remote computation. For instance, Jini allows an application to discover the interface of a resource through a directory service, transparently download a stub, and remotely compute with that resource.

All of these systems provide users some control over how code and computation should be distributed. However, unlike MAGE, the support here is primarily for distributing code. MAGE, on the other hand, integrates the notion of computation and distribution through the notion of mobility attribute, thereby providing a more general and unified framework.

6.2 Mobile agent based approaches

Examples of early work on mobility of programs (and objects) through a language's runtime system are Emerald [15], Hermes [8], and COOL [14]. The Emerald runtime provides an abstraction of a single address space over multiple hosts connected through a local area network. One of the novel components of the Emerald system is its ability to directly map objects into a local address space, unmap it, and then re-map it at a remote node. Emerald also provides language support for explicitly migrating objects. Hermes is a runtime system that is independent of applications, operating systems, or programming language. Mobility in both Emerald and DOWL [3] is achieved by associating location properties with objects. We, on the other hand, focus on associating mobility properties with computations.

Recently, several programming languages such as Telescript [30], AgentTCL [17], Aglet [18], Mole [25], Ara [21], Ajanta [27] and Sumatra [22] have been designed to support mobility of programs over the wide area network. We can classify [16] these systems into two: In the first, migration of both program and execution states is supported. Examples of systems that support this are Sumatra and Telescript. In the second, the notion of mobility is achieved by imposing constraints on how and when programs can migrate. These restrictions arise because the JVM does not export an application's execution memory segments. For this reason, the mobile agent model in MAGE also uses weak mobility for migrating active objects.

Our work differs from MA-based approaches in how we look at mobility. In our model, the basis for migration is not only a program, but any of its components. Further, the mobility properties of the components can be changed on the fly to suit the underlying conditions.

Several programming languages allow a combination of the different mobility models. For instance, Active Names [28] allow one to associate a resource name with specific programs in different name spaces. Further, these programs can be downloaded and composed to provide extensibility and flexibility. Similarly, Stratosphere allows one to use the different mobility models for writing distributed programs. The Stratosphere programming model is the closest to our model. Our approach differs from the Stratosphere [31] programming model in the treatment of mobility. In our approach, mobility is defined as a property of a computation and can be modeled and manipulated directly through mobility attributes.

7 Conclusion

Currently, MAGE trusts its constituent servers. We are exploring a version of MAGE that runs on and scales to WANs consisting of large, heterogenous networks, fragmented into competing and disjoint administrative domains, each with different services, resources and security needs — in short, the Internet. We also are working on adding access control and resource allocation models to MAGE.

MAGE has inherited RMI's reliance on static information shared between clients and servers. This is not surprising, since RMI is MAGE's foundational substructure. Specifically, MAGE requires that mobile objects and their clients share the name of the mobile object's origin server, an interface to the mobile object and the mobile object's name as bound in the MAGE registry. Re-implementation using Jini would directly and simply solve this problem.

MAGE's raison d'etre is that computation and resources must be dynamically collocated 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 supports mobile objects, the namespaces in which they execute, mechanisms by which objects can move and various services that support these tasks.

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.

References

[1] Blackstone Technology Group. `www.computefarm.com`.

[2] Java Remote Method Interface (RMI). `java.sun.com/products/jdk/rmi/index`.

[3] B. Achauer. The DOWL Distributed Object-Oriented Language. *CACM*, 36(9):48–55, September 1993.

[4] K. Arnold and J. Gosling. *The Java Programming Language Third Edition*. Addison Wesley, 2000.

[5] Y. Artsy and R. Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, pages 47–56, Sept. 1989.

[6] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret. The World-Wide Web. *CACM*, 37(8):76–82, August 1994.

[7] A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, February 1984. 2(1).

[8] A. Black and Y. Artsy. Implementing Location Independent Invocation. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):107–119, 1990.

[9] A. Carzaniga, P. Pietro, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Int'l Conference on Software Engineering*, pages 22–32, Boston, MA, May 1997.

[10] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? Technical report, IBM T.J. Watson Research Center, 1995.

[11] F. Douglis and J. Ousterhout. Transparent Process migration: Design alternatives and the Sprite Implementation. *Software Practice and Experience*, 21(8):757–785, 1991.

[12] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[13] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proc. of the 1999 Usenix ATC*.

[14] S. Habert and L. Mosseri. COOL: Kernel Support for Object-Oriented Environments. In *Proceedings of the ECOOP/OOPSLA*, pages 269–277, 1990.

[15] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[16] J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet*, pages 21–30, July-August 1997.

[17] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent TCL: Targeting the Needs of Mobile Computers. *IEEE Internet Computing*, 1(4), July/August 1997.

[18] D. Lange, M. Oshima, G. Karjoth, and K. Kosaka. Aglets: Programming Mobile Agents in Java. In *Proc. of the Worldwide Computing and Its Applications*, pages 253–266, 1997.

[19] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor — a hunter of idle workstations. In *Proc. of the 8th Int'l Conference on Distributed Computing Systems*, June 1988.

[20] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *Proc. of the ACM 1999 conference on Java Grande*, 1999.

[21] H. Peine and T. Stoplmann. The Architecture of the Ara Platform for Mobile Agents. In *Proceedings of the First International Workshop on Mobile Agents'97*, 1997.

[22] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz. Network-aware Mobile Programs. In *Proceedings of the Annual Usenix'97 Conference*, 1997.

[23] J. Siegel. *CORBA: Fundamental and Programming*. Wiley, 1996.

[24] J. Stamos and D. Gifford. Remote Evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[25] M. Straber, J. Baumann, and F. Hohl. Mole: A Java based Mobile Agent System. In *Proceedings of the 2nd ECOOP Workshop on Mobile Object System*, 1997.

[26] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[27] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh. Mobile Agent Programming in Ajanta. In *Proceedings of the 19th International Confernce on Distributed Computing Systems (ICDCS)*, 1999.

[28] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proc. of the 2nd USENIX Symposium on Internet Technologies and Systems*, pages 151–164, 1999.

[29] J. Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM,*, 42(7):76–82, 1999.

[30] J. White. Telescript Technology: The Foundation for the Electronic Marketplace. General Magic Inc. White Paper, 1994. http://www.magic.com.

[31] D. Wu, D. Agrawal, and A. E. Abbadi. StratOSphere: Mobile Processing of Distributed Objects in Java. *The fourth annual ACM/IEEE international conference on Mobile computing and networking*, pages 121–132, 1998.