J-Orchestra: Automatic Java Application Partitioning

Eli Tilevich and Yannis Smaragdakis
Center for Experimental Research in Comp. Science (CERCS), College of Computing
Georgia Institute of Technology, Atlanta, GA 30332
{tilevich, yannis}@cc.gatech.edu
http://j-orchestra.org

Abstract. J-Orchestra is an automatic partitioning system for Java programs. J-Orchestra takes as input Java applications in bytecode format and transforms them into distributed applications, running on distinct Java Virtual Machines. To accomplish such automatic partitioning, J-Orchestra uses bytecode rewriting to substitute method calls with remote method calls, direct object references with proxy references, etc. Using J-Orchestra does not require great sophistication in distributed system methodology—the user only has to specify the network location of various hardware and software resources and their corresponding application classes. J-Orchestra has significant generality, flexibility, and degree of automation advantages compared to previous work on automatic partitioning. For instance, J-Orchestra can correctly partition almost any pure Java program, allowing any application object to be placed on any machine, regardless of how application objects access each other and Java system objects. This power is due to the novel way that J-Orchestra deals with unmodifiable code (e.g., native code in the Java system classes). Additionally, J-Orchestra offers support for object migration and run-time optimizations, like the lazy creation of distributed objects.

We have used J-Orchestra to successfully partition several realistic applications including a command line shell, a ray tracer, and several applications with native dependencies (sound, graphics).

1 Introduction

Application partitioning is the task of breaking up the functionality of an application into distinct entities that can operate independently, usually in a distributed setting. Application partitioning has been advocated strongly in the computing press [11] as a way to use resources efficiently. Traditional partitioning entails re-coding the application functionality to use a middleware mechanism for communication between the different entities. In this paper, we present an *automatic partitioning system* for Java applications. Our system, called J-Orchestra, utilizes compiler technology to partition existing applications without manual editing of the application source code.

Automatic partitioning aims to satisfy functional constraints (e.g., resource availability). For instance, an application may be getting input from sensors, storing it in a database, processing it, and presenting the results on a graphical screen. All four hardware resources (sensors, database, fast processor, graphical screen) may be on different machines. Indeed, the configuration may change several times in the lifetime of the application. Automatic partitioning can accommodate such requirements without needing to hand-modify the application source code. Thus, automatic partitioning is a sophisticated alternative to input-output re-direction protocols (Java servlets, telnet, X-Windows [15]). Automatic partitioning can do whatever these technologies do, with the additional advantage that the partitioning of the application is completely flexible—different parts of the application can run on different machines in order to minimize network traffic or reduce server load. For instance, instead of using X-Windows to send graphics over the network, one can keep the code generating the graphics on the same site as the graphics hardware.

J-Orchestra operates at the Java bytecode level and rewrites the application code to replace local data exchange (function calls, data sharing through pointers) with remote communication (remote function calls through Java RMI [18], indirect pointers to mobile objects). The resulting application is guaranteed to have the same behavior as the original one (with a few, well-identified exceptions). J-Orchestra receives input from the user specifying the network locations of various hardware and software resources and the code using them directly. A separate profiling phase and static analysis are used to automatically compute a partitioning that minimizes network traffic.

Although the significance of J-Orchestra may appear Java-specific, there is a general conceptual problem that J-Orchestra is the first system to solve. This is the problem of supporting transparent reference indirection in the presence of unmodifiable code. More specifically, J-Orchestra is one of many systems that work by changing all direct references to objects into indirect references (i.e., references to proxy objects). This approach is hard to implement transparently when the program consists partly of unmodifiable code. We show that J-Orchestra can “work around” unmodifiable code, ensuring that it is clearly isolated from modifiable code by dynamically “wrapping” direct references to make them indirect (and vice versa), when the references are passed from unmodifiable to modifiable code (and vice versa).

The result of solving the problems with unmodifiable code is that J-Orchestra is the first automatic partitioning system that imposes no partitioning constraints on application code. (We make a clear distinction between “automatic partitioning” systems and general “Distributed Shared Memory” mechanisms in our related work discussion.) Unlike previous systems (e.g., Addistant [19]—the most mature and closest alternative to J-Orchestra in the design space) J-Orchestra can partition any Java application, allowing any *application object* to be placed on any machine, regardless of how application objects interact among them and with system objects. Any *system object* can be remotely accessed from anywhere in the network, although it has to be co-located with system objects that may potentially reference it. (The terms “application” and “system” objects roughly correspond to instances of regular classes of a Java application, and of Java system classes with native dependencies, respectively.)

In this paper, we present the main elements of the J-Orchestra rewrite engine. We describe the J-Orchestra rewrite algorithm, discuss its power and detail how J-Orchestra deals with various features of the Java language. Finally, we examine some J-Orchestra optimizations and present performance measurements that demonstrate the advantage of J-Orchestra over input/output redirection with X-Windows.

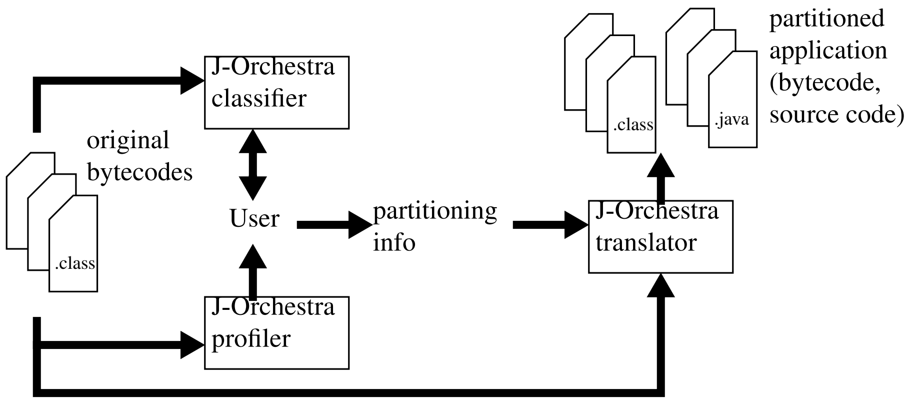

Fig. 1. An overview of the J-Orchestra partitioning process

2 System Overview

We will give here a high-level overview of the operation of J-Orchestra from the perspective of a user (see Fig. 1). Many important details are elided—they will be added in the next few sections. Some low-level details will be left unspecified as they may soon change. For instance, currently the interaction of the user and the J-Orchestra system is done using scripts and XML-based configuration files, but a complete GUI that will hide many of these details will be available by the time of publication.

The user interaction with the J-Orchestra system consists of specifying the mobility properties and location of application objects. J-Orchestra converts all objects of an application into *remote-capable* objects—i.e., objects that can be accessed from a remote site. Remote-capable objects can be either *anchored* (i.e., they cannot move from their location) or *mobile* (i.e., they can migrate at will). For every class in the original application, or Java system class potentially used by application code, the user can specify whether the class instances will be mobile or anchored. For mobile classes, the user needs to also describe a migration policy—a specification of when the objects should migrate and how. For anchored classes, the user needs to specify their location. Using this input, the *J-Orchestra translator* modifies the original application and system bytecode, creates new binary packages, produces source code for helper classes (proxies, etc.), compiles that source code, and creates the final distributed application.

Specifying the properties (anchored or mobile, migration policy, etc.) of an application or system class is not a trivial task. A wrong choice may yield an inefficient or incorrect distributed application. For instance, many system classes have interdependencies so that they all need to be anchored on the same site for the application to work correctly. To ensure a correct and efficient partitioning, J-Orchestra offers two tools: a *profiler* and a *classifier* (Fig. 1).

The profiler is the simpler of the two: it reports to the user statistics on the interdependencies of various classes based on (off-line) profiled runs of the application. With this information, the user can decide which classes should be anchored together and where. J-Orchestra includes heuristics that compute a good partitioning based on profiling data—the user can run these heuristics and override the result at will.

The J-Orchestra classification algorithm is responsible for ensuring the correctness of the user-chosen partitioning. The classifier analyzes classes to find any dependencies that can prevent them from being fully mobile. One of the novelties of J-Orchestra is that regular application classes can almost always be mobile. Nevertheless, Java system classes, as well as some kinds of application classes, may have dependencies that force them to be anchored. As discussed in Section 4, example dependencies include an implementation in native (i.e., platform-specific) code, possible access to instances of the class from native code, inheriting from a class that is implemented in native code, etc. The interaction of the user with the classifier is simple: the classifier takes one or more classes and their desired locations as input and computes whether they can be mobile and, if not, whether the suggested locations are legal and what other classes should be co-anchored on the same sites. The user interacts with the classifier until all system classes have been anchored correctly.

In the next sections, we describe the J-Orchestra classification and translation algorithms in detail.

3 Rewrite Strategy Overview

3.1 Main Insights

J-Orchestra creates an abstraction of shared memory by allowing references to objects on remote JVMs. That is, the J-Orchestra rewrite converts all references in the original application into *indirect references*—i.e., references to *proxy objects*. The proxy object hides the details of whether the actual object is local or remote. If remote methods need to be invoked, the proxy object will be responsible for propagating the method call over the network. Turning every reference into an indirect reference implies several changes to application code: for instance, all `new` statements have to be rewritten to first create a proxy object and return it, an object has to be prevented from passing direct references to itself (`this`) to other objects, etc. If other objects need to refer to data fields of a rewritten object directly, the code needs to be rewritten to invoke accessor and mutator methods, instead. Such methods are generated automatically for every piece of data in application classes. For instance, if the original application code tried to increment a field of a potentially remote object directly, as in `o1.a_field++`, the code will have to change into `o1.set_a_field(o1.get_a_field()+1)`. (This rewrite will actually occur at the bytecode level.)

The above indirect reference techniques are not novel (e.g., see JavaParty [8], as well as the implementation of middleware like Java RMI [18]). The problem with indirect reference techniques, however, is that they do not work well when the remote object and the client objects are implemented in *unmodifi able code* Typically, code is unmodifiable because it is native code—i.e., code in platform specific binary form. For instance, the implementation of many Java system classes falls in this category. Unmodifiable code may be pre-compiled to refer directly to another object's fields, thus rendering the proxy indirection invalid. One of the major novel elements of J-Orchestra is the use of indirect reference techniques even in the presence of unmodifiable code.

3.2 Handling Unmodifiable Code

To see the issues involved, let us examine some possible approaches to dealing with unmodifiable code. We will restrict our attention to Java but the problem (and our solution) is general: pre-compiled native code that accesses the object layout directly will cause problems to indirect reference approaches in any setting.

- If the client code (i.e., holder of a reference) of a remote object is not modifiable, but the code of the remote object is modifiable, then we can use "name indirection": the proxy class can assume the name of the original remote class, and the remote class can be renamed. This is the "replace" approach of the Addistant system [19]. The problem is that the client may expect to access fields of the remote object directly. In this case, the approach breaks.
- If the client code (i.e., holder of a reference) of a remote object is modifiable but the code of the remote object is not, then we can change all clients to refer to the proxy. This is the "rename" approach of the Addistant system. This case does not present any problems, but note that the Addistant approach is "all-or-none". *All* clients of the unmodifiable class must be modifiable, or references cannot be freely passed around (since one client will refer to a proxy object and another to the object directly).
- If the client code (i.e., holder of a reference) of a remote object is not modifiable and the code of the remote object is also not modifiable, no solution exists. There is no way to replace direct references with indirect references. Nevertheless, the key observation is that unmodifiable clients can refer to the remote object directly, while modifiable clients refer to it indirectly. In this way, although unmodifiable objects cannot be placed on different network sites when they reference each other, modifiable objects can be on a different site than the unmodifiable objects that they reference. *This is the approach that J-Orchestra follows.* A direct consequence is that (unlike the Addistant rewrite) the semantics of the application does not affect its ability to be partitioned. An application object (instance of a modifiable class) can be placed anywhere on the network, regardless of which Java system objects it accesses and how.

  For this approach to work, it must be possible to create an indirect reference from a direct one and vice versa, at application run-time. The reason is that references can be passed from modifiable to unmodifiable code and vice versa by using them as arguments or results of a method call. Fortunately, this conversion is easy to handle since all method calls are done through proxies. Proxies for unmodifiable classes are the only way to refer to unmodifiable objects from modifiable code. When a method of such a proxy is called, the reference arguments need to be *unwrapped* before the call is propagated to the target object. Unwrapping refers to creating a direct reference from an indirect one. Similarly, when a method of such a proxy returns a reference, that reference needs to be *wrapped*: a new indirect reference (i.e., reference to a proxy object) is created and returned instead.

A consequence of the J-Orchestra rewrite algorithm is that is supports object mobility. If an object can only be referenced through proxies, then its location can change transparently at run-time. Thus, for instance, regular application objects in a "pure Java" application can migrate freely to other sites during application execution. (An exception is the case of application classes that extend system classes other than the default subtyping root, `java.lang.Object`—see Section 4.2.2.) In contrast, many instances of Java system classes are remotely accessible but typically cannot migrate, as they may be accessed directly by native code.

4 Rewrite Mechanism

In this section, we discuss in concrete detail the J-Orchestra rewrite model. As described in Section 2, J-Orchestra distinguishes between anchored and mobile classes. Unmodifiable classes have to be anchored, but modifiable classes can be either anchored or mobile. The J-Orchestra mechanisms of *classifi cation*and *translation* are entirely separate. The purpose of the J-Orchestra classifier is to determine whether an object should be anchored (and where) or mobile. This algorithm could change in the future, while the translation mechanism for mobile classes, anchored unmodifiable classes, and anchored modifiable classes stays the same. Similarly, the translation mechanism for the three categories of classes can change, even if the way we determine the category of a class remains the same.

In the following sections, we will blur the distinction between classes and their instances when the meaning is clear from context. For instance, we write "class `A` refers to class `B`" to mean that an instance of `A` may hold a reference to an instance of `B`.

4.1 Classification

Classes may have to be anchored if they have native methods or if they may potentially be manipulated by native code. For example, J-Orchestra's rewrite engine deems `java.lang.ThreadGroup` anchored because a reference to a `ThreadGroup` can be passed to the constructor of class `java.lang.Thread`, which has native methods.

Fig. 2 shows the different categories in which classes are classified by J-Orchestra. The classification criteria for the vast majority of classes can be summarized as follows. (Some exceptions will be discussed individually.)

- *Anchored Unmodifi able Classes* A class `C` is anchored unmodifiable if it has native methods, or references to `C` objects can be passed between modifiable code and an anchored unmodifiable class `U`. In the latter case, classes `C` and `U` need to be anchored on the same network site.

  For simplicity, we assume in this paper that the application to be partitioned is written in pure Java (i.e., the only access to native code is inside Java system

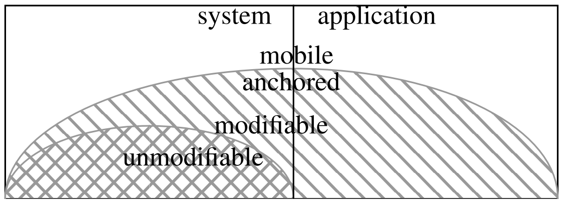

Fig. 2. The possible categories of classes. Unmodifiable classes need to be anchored, but both system and application classes can be modifiable and even modifiable classes may be anchored (by need or by choice). For simplicity, we ignore the possibility of unmodifiable application classes.

classes). Thus, application classes are modifiable—only system classes can be unmodifiable. This is the standard usage scenario for J-Orchestra. It is straightforward to generalize our observations to applications that include native code.[1]

- *Anchored Modifi able Classes*: A class is anchored modifiable if it is a modifiable application class that extends an anchored unmodifiable class (other than `java.lang.Object`). These classes need to be anchored on the same site as their superclasses.

  Additionally, a modifiable class may be anchored by choice (see Section 5.1).

- *Mobile Classes*: Mobile classes are all classes that do not fall in either of the above two categories. All classes in a pure Java application that do not extend system classes are mobile. Note, however, that Java system classes can also be mobile, as long as they do not call native code and they cannot be passed to/from anchored system classes. In this case, instances of the system class are used entirely in "application space" and are never passed to unmodifiable code. The implementation of such classes can be replicated in a different (non-system) package and application code can be rewritten to refer to the new class. The system class can be treated exactly like a regular application class using this approach.

Note that static inspection can conservatively guarantee that references to a system class `C` never cross the system/application boundary. As long as no references to `C` or its superclasses (other than `java.lang.Object`) or to arrays of these types appear in the signatures of methods in anchored system classes, it is safe to create a mobile "application-only" version. (Interface access or access through or `java.lang.Object` references is safe—a proxy object is indistinguishable from the original object in these cases.) As a consequence, the categorization of system classes into mobile and anchored is robust with respect to future changes in the implementation of Java library classes—the partitioning remains valid as long as the interfaces are guaranteed to stay the same.

1. If the application includes native code, our guarantees will need to be adjusted. For an extreme example, if native code in a single method accesses fields of all application classes directly, then no partitioning can be done, since all application classes will need to be anchored on the same site.

```
compute_co-anchored (A) {
  AS := set of all mutable system classes and all array types
  A := A ∪ Superclasses(A) ∪ Subclasses(A)
  do {
    AS := AS - A
    AArg := MethodArguments(A)
    AArg := AArg ∪ Superclasses(AArg) ∪ Subclasses(AArg) ∪ Constituents(AArg)
    ArgS := AS ∩ AArg
    A := A ∪ ArgS
  } while (ArgS ≠ ∅)
  return A
}
```

Fig. 3. J-Orchestra algorithm to compute anchored unmodifi able classes

More concretely, the J-Orchestra algorithm to compute anchored unmodifiable classes can be seen in set pseudo-code notation in Fig. 3. This algorithm finds the classes that need to be anchored on the same site as any one of the classes of an initial set *A*. By changing the input set *A*, we adapt this algorithm for several different purposes throughout J-Orchestra. The auxiliary set routines used in this algorithm are defined as follows: *Super(Sub)classes(X)* returns the set of all super(sub)classes of classes in set *X*; *MethodArguments(X)* returns the set of all argument and return types of all methods of all classes in *X*; *Constituents(X)* returns the set of all constituent types of all array types in *X*. For instance, an array type `T[][]` has constituent types `T[]` and `T`.

We should mention that, anchoring system classes together with other related system classes typically does not inhibit the meaningful partitioning of system resources. For instance, we have used J-Orchestra to partition several applications so that the graphics display on one machine, while disk processing, sound output, keyboard input, etc. are provided on remote computers. This is possible because classes within the same Java system package reference mostly each other and very rarely system classes from other packages. This property means that anchoring group boundaries commonly coincide with package boundaries. For example, all the classes from the `java.awt` package can be anchored on the same machine that handles the user interface part of an application. This arrangement allows anchored system classes to access each other directly while being remotely accessible by application classes through proxies.

As an advanced technical note, we should mention that less conservative classification rules can also be applied to guarantee that more system classes can be made mobile. For instance, if a system class never accesses native code, never has its fields directly referenced by other system classes (i.e., all access is through methods), and its instances are passed from application classes to system classes but not the other way, then the class can be mobile by using a “subtype” approach: a subtype of the system class can be created in an application package. The subtype is used as a proxy—none of its original data fields are used. Nevertheless, the subtype object can be safely passed to system code when the supertype is expected. The subtype object itself propagates all method calls to an actual mobile object. This technique is applicable as long as the original system class is not `final`. We already use this technique in J-Orchestra but not automatically—manual intervention is required to enable this transformation on a case-by-case basis when it seems warranted. A good example is the `java.lang.Vector` class. Vectors are used very often to pass data around and it would be bad for performance to restrict their mobility: vectors should migrate where they are needed. Nevertheless, many graphical applications pass vectors to Swing library anchored system classes—e.g., the `javax.swing.table.DefaultTableM-odel` class has methods that expect vectors. All the aforementioned conditions are true for vectors: the `Vector` class has no native methods, classes in the Swing library do not access fields of vector objects directly (only through methods), and vectors are only passed from application to system code, but not the other way. Therefore, `Vector` can be safely turned into a mobile class in this case.

For a more accurate determination of whether system classes can be made mobile, data flow analysis should be employed. In this way, it can be determined more accurately whether instances of a class flow from application code to system code. So far, we have not needed to exploit such techniques in J-Orchestra—the type system has been a powerful enough ally in our effort to determine which objects can be made mobile.

4.2 Translation

4.2.1 Anchored Unmodifiable (System) Classes

J-Orchestra does not modify anchored system classes but produces two supporting classes per anchored system class. These are a proxy class and a *remote application-system translator* (or just *application-system translator*). A proxy exposes the services of its anchored class to regular application classes. A remote application-system translator enables remote execution and handles the translation of object parameters between the application and system layers.[2] Both proxy classes and remote application-system translator classes are produced in source code form and translated using a regular Java compiler. We will now examine each of these supporting classes in detail.

A proxy is a front-end class that exposes the method interface of the original system class. It would be impossible to put a proxy into the same package as the original system class: system classes reside in system packages that J-Orchestra does not modify. Instead, proxies are placed in a different package and have no relationship to their system classes. Proxy naming/package hierarchies are isomorphic to their corresponding system classes. For example, a proxy for `java.lang.Thread` is called

2. The existence of a separate application-system translator is an RMI-specific implementation detail—under different middleware, the translator functionality could be folded inside the proxy. Under RMI, classes need to explicitly declare that they are remotely accessible (e.g., by inheriting from class `UnicastRemoteObject`). Therefore, unmodifiable system classes cannot be made remotely accessible, but their translator can. Separate application-system translators simplify our implementation because system classes wrapped with an application-system translator can be treated the same as application classes.

anchored.java.lang.Thread. To make remote execution possible, all modifiable classes that reference the original system class have to now reference the proxy class instead. This is accomplished by consistently changing the constant pools of all the modifiable binary class files. The following example demonstrates the effect of those changes as if they were done on the source code level for clarity reasons.

```
//Original code: client of java.lang.Thread
java.lang.Thread t = new java.lang.Thread (...);
void f (java.lang.Thread t){ t.start (); }

//Modified code
anchored.java.lang.Thread t =
  new anchored.java.lang.Thread (...);
void f (anchored.java.lang.Thread t) { t.start ();              }
```

All the object parameters to the methods of a proxy are either immutable classes such as java.lang.String or other proxies. The rewrite strategy ensures that proxies for anchored system classes do not reference any other anchored system classes directly but rather through proxies.

The only data member of an anchored system proxy is an interface reference to the remote application-system translator class. A typical proxy method delegates execution by calling an appropriate method in the remote instance member and then handles possible remote exceptions. For instance, the setPriority method for the proxy of java.lang.Thread is:

```
public final void setPriority(int arg0){
  try { _remoteRef.setPriority (arg0); }
  catch (RemoteException e) {        e.printStackTrace ();       }
}
```

The _remoteRef member variable can point to either the remote application-system translator class itself or its RMI stub. In the first case, all method invocations will be local. Invocations made through RMI stubs go over the network, eventually getting handled by the system object on a remote site.

Application-system translators enable remote invocation by extending java.rmi.server.UnicastRemoteObject.[3] Additionally, they handle the translation of proxy parameters between the application and user layers. Before a proxy reference is passed to a method in a system class, it needs to be unwrapped. Unwrapping is the operation of extracting the original system object pointed to by a proxy. If a system class returns an instance of another system class as the result of a method call, then that instance needs to be wrapped before it is passed to the application layer. Using wrap-

3. While this is not the only way to achieve remote semantics (a class can simply implement java.rmi.Remote and then use javax.rmi.PortableRemoteObject.export() to export objects later on), UnicastRemoteObject provides several important services (e.g., identity—see Section 4.3.6), and so far we have chosen to avoid re-implementing them.

ping, J-Orchestra manages to be oblivious to the way objects are created. Even if system objects are created by unmodifiable code, they can be used by regular application classes: they just need to be wrapped as soon as they are about to be referenced by application code.

The following example demonstrates how "wrapping-unwrapping" works in methods `setForeground` and `getForeground` of the application-system translator for `java.awt.Component`.

```
public void setForeground (anchored.java.awt.Color arg0) {
  _localClassRef.setForeground
    ((java.awt.Color)Anchored.unwrapSysObj (arg0));
}

public anchored.java.awt.Color getForeground () {
  return
    (anchored.java.awt.Color)
      Anchored.wrapSysObj(_localClassRef.getForeground());
}
```

`_localClassRef` points to an instance of the original system class (`java.awt.Component`) that handles all method calls made through the application-system translator.

4.2.2 Anchored Modifiable Classes

Anchored modifiable classes are the application classes that inherit from anchored system classes or any otherwise modifiable class that is anchored by choice. Anchored modifiable classes are handled with a translation that is identical to the one for anchored unmodifiable classes, except for one aspect. The defining distinction between unmodifiable and modifiable anchored classes is that the latter can be changed so that, if they access other classes' fields directly, such accesses can be replaced with calls to accessor and mutator methods. In this way, other classes referenced by anchored modifiable classes do not need to be anchored.

4.2.3 Mobile Classes.

Mobile classes are able to migrate to various network sites during the run of a program. The migration currently supported by J-Orchestra is *synchronous*: objects migrate in response to run-time events, such as passing a mobile object as a parameter to a remote method. Migration allows us to exploit data locality in an application. For instance, when a remote method call occurs, it can be advantageous to have a mobile object parameter move temporarily or permanently to the callee's network site. All standard object mobility semantics (e.g., call-by-visit, call-by-move [10]) can be supported in an application rewritten by J-Orchestra.

J-Orchestra translates mobile classes in the original application (and the replicated mobile system classes) into a *proxy class* and a *remote class*. Proxy classes are created in source code form, while remote classes are produced by bytecode rewriting of the original mobile class. Proxies for mobile classes are very similar to the ones for anchored classes. The only difference is that *a mobile proxy assumes the exact name and method interface of the original class*. J-Orchestra adds an "__remote" suffix to the original class name. The clients of a mobile class access its proxy in exactly the same way as they used to access the original class.

Mobile class proxies mimic the inheritance structure of their original classes. The remote semantics is achieved by changing the superclass of the base (topmost) proxy from `java.lang.Object` to `java.rmi.server.UnicastRemoteObject`.

The example below summarizes the rewrite in source code form (although in reality the original class and the remote class only exist in bytecode form).

```
//Original class declaration
class A extends B implements I {...}

//Proxy class declaration.
//B or one of its ancestors inherit from UnicastRemoteObject
class A extends B implements I, Proxy { ... }

//Remote class declaration
//body of A__remote is same as body of original A
class A__remote extends B__remote implements I, Remote {...}
```

Some care needs to be taken during binary modification of a class, to ensure that the types expected match the ones actually used. For instance, the name of a class `A` needs to change to `A__remote`, but most references to type `A` (e.g., as the type of a method parameter) need to continue referring to `A`—the proxy type is the right type for references to `A` objects in the rewritten application.

4.3 Handling of Java Language Features

In this section, we describe how J-Orchestra handles various Java language features. Some parts of the translation (e.g., that of static methods) are straightforward and only add engineering complexity. Handling other elements (e.g., arrays), however, is far from trivial. Some of the techniques described here are similar to the ones used by JavaParty (but JavaParty operates at the source code level while J-Orchestra is a bytecode translator).

Maintaining exactly the local execution semantics is not always possible or efficient. We will identify the few features for which J-Orchestra will not guarantee, by need or by choice, that the partitioned application will behave exactly like the original one.

4.3.1 Static Methods and Fields

J-Orchestra has to handle remote execution of static methods. This also takes care of remote access to static fields: just like with member fields, J-Orchestra replaces all direct accesses to static fields of other classes with calls to accessor and mutator methods. In order to be able to handle remote execution of static methods, J-Orchestra creates static delegator classes for every original class that has any static methods. Static delegators extend `java.rmi.server.UnicastRemoteObject` and define all the static methods declared in the original class.

```
//Original class
class A {
  static void foo (String s) {...}
  static int bar () {...}
}

//Static Delegator for A—runs on a remote site
class A__StaticDelegator
 extends java.rmi.server.UnicastRemoteObject {
  void foo (String s) { A__remote.foo (s); }
  int bar () { return A__remote.bar (); }
}
```

For optimization purposes, a static delegator for a class gets created only in response to calling any of the static methods in the proxy class. If no static method of a class is ever called during a particular execution scenario, the static delegator for that class is never created. Once created, the static delegator or its RMI stub is stored in a member field of the class's proxy and is reused for all subsequent static method invocations.

A static delegator for a class shares the mobility properties of the class itself. While a static delegator for an anchored class must be co-anchored on the same site, the static delegator of a mobile class can potentially migrate at will, irrespective of the locations of the existing objects of its class type.

4.3.2 Inheritance

Proxies, remote application-system translator classes, and remote classes all mimic the inheritance/subtyping hierarchy of their corresponding original classes. Replacing direct references with references to proxies preserves the original execution semantics: a proxy can be used when a supertype instance is expected. Since it is not known which particular proxy is going to be used to invoke a method, only the base class contains the interface reference that is used for method delegation. This field is accessible to all the subclasses' proxies by having the `protected` access modifier.

4.3.3 Object Creation

Creating objects remotely is a necessary functionality for every distributed object system. J-Orchestra proxies' constructors work differently from other methods in order to implement distribution policies (i.e., create various objects on given network sites). First, a proxy constructor calls a special-purpose do-nothing constructor in its super class to avoid the regular object creation sequence. A proxy constructor creates objects using the services of the *object factory*. J-Orchestra's object factory is an RMI service running on every network node where the partitioned application operates. Every object factory is parameterized with configuration files specifying a symbolic location of every class in the application and the URLs of other object factories. Every *object factory client* keeps remote references to all the object factories in the system. Object factory clients determine object locations, handle remote object creations, and maintain various mappings between the created objects and their proxies. The following example shows a portion of the constructor code in a proxy class `A`.

```
public  A () {
  //call super do-nothing constructor
  super ((BogusConstructorArg)null);

  //check if we are already initialized or are
  //called from a subclass
  if ((null != _remoteRef) || (!getClass ().equals (A.class)))
    return;
  ...
  //Call ObjectFactory
  try { _remoteRef = (A) ObjectFactory.createObject("A"); }
  catch (RemoteException e) { ... }
}
```

4.3.4 Arrays

Handling arrays is interesting from a language standpoint because they are the only native generic type in Java. Conceptually, arrays are very similar to objects. For instance, arrays are subclasses of `java.lang.Object`. An array can be thought of as a class that supports the operations "store" and "load". Arrays require special treatment because, just like objects, they are mutable and can be aliased: changes made through one array reference have to be visible to all other references to the same array.

J-Orchestra treats arrays very similarly to objects, although at the concrete level the translation is different. All arrays are wrapped into special *array front-end* classes for reference by the application. Application classes are modified to replace array accesses with calls to the "store" and "load" methods of an array front-end. The front-end is responsible for performing the appropriate operations on the array itself. If the array type is mobile, then the array front-end is treated exactly like a regular mobile class (i.e., a proxy is created for it). If, however, the array type is anchored, the front-end has a dual role. It also serves as a system/application translator and automatically wraps and unwraps the elements inserted into arrays. For instance, the front-end for an anchored array of `java.lang.Thread` objects is responsible for wrapping the thread objects when they are retrieved by application code and unwrapping them when they are stored. This front-end class is shown here:

```
class java_lang_Thread_FrontEnd {
  java.lang.Thread []_array;

  anchored.java.lang.Thread aaload(int location) {
    return   (anchored.java.lang.Thread)
             Anchored.wrap (_array[location]);
  }

  void aastore (int location, anchored.java.lang.Thread elem) {
    _array[location] = (java.lang.Thread)Anchored.unwrap (elem);
  }
}
```

It is worth noting that the same "wrapping/unwrapping" needs to be performed for multidimensional anchored arrays. For instance, if a two dimensional array of integers is anchored, then before each of its constituent arrays is retrieved, it needs to be wrapped in a front-end for one dimensional integer arrays. The code fragment below (a slight simplification of the actual J-Orchestra generated code) shows this transformation.

```
class Int2FrontEnd {
  int [][] _array;
  Int2FrontEnd (int[][]array) {_array = array;}
  int [][] get_array () { return _array; }

  IntFrontEnd aaload (int location) {
    return new IntFrontEnd(_array[location]);
  }
  void aastore (int location, IntFrontEnd value) {
    _array[location] = value.get_array ();
  }
}
```

Determining whether an array needs to be anchored or can be mobile is an interesting problem. Although arrays are implemented in native code, we can safely assume that they do not capture system-specific state and that they never directly access fields of the arguments to their "store" and "load" methods, as they have no knowledge of the types of the array elements. Therefore, arrays can be made mobile, unless they are passed between application code and system code. Note that this means that an array of objects of class C can be mobile even when class C is anchored—C objects may cross the application/system boundary, but as long as *arrays* of C objects do not cross it, these arrays can be made mobile.

Nevertheless, the usual type-based anchored/mobile classification mechanism of J-Orchestra can be too restrictive when applied to arrays. Recall that according to the J-Orchestra classification, if a reference to a certain type can cross the system/application boundary, then all references to this type are made anchored. Some of the consequences of this approach are: a) if a multidimensional array is anchored, then every array of the same or lower dimension and the same element type also needs to be anchored on the same site; b) if an array of C objects is anchored to a site, then all arrays of subclass objects of the same dimension need to be anchored on the same site. For primitive types (`int`, `float`, etc.) the problem becomes even more intense. The problem is that the J-Orchestra classification algorithm is type based and primitive array types are anonymous types. The same type, e.g., `int[]`, can be used for very different purposes, but currently J-Orchestra can only be conservative due to lack of data flow information. For instance, any application that passes an integer array to an anchored system class will have to treat all its integer arrays (of the same or lower dimension) as anchored *on the same site*! This restriction may even hinder the ability to safely place different Java system classes on different network sites. If two entirely unconnected system packages both exchange arrays of integers with some application's code, then both packages have to be placed on the same machine, because of the possibility that they both refer to the same array.

In the future, we plan to explore more sophisticated classification algorithms to automatically ensure that arrays can be mobile safely. For now, manual intervention is the only way to circumvent the rigidness of the J-Orchestra classification. Unfortunately, safety is not automatically ensured in this case. Note that the only problem concerns the read-write use of arrays: if arrays are only written by application code and read by system code (or vice-versa), they can safely be made mobile. Fortunately, this is the common for arrays shared between application and system code, but J-Orchestra cannot know this without manual hints.

We have partitioned several Java applications using J-Orchestra without ever needing to exercise manual control in order to overcome array classification problems.

4.3.5 "this"

Under the J-Orchestra rewrite, an object can refer to its own methods and variables directly. That is, no proxy indirection overhead is imposed for access to methods through the `this` reference. Nevertheless, this means that J-Orchestra has to treat explicit uses of `this` specially. Recall that remote objects are generated by changing the name of the original class at the bytecode level. When the name of a class changes so does the type of all of its explicit `this` references. Consider the following example showing the problem if no special care is taken:

```
//original code
class A {   void foo (B b) {      b.baz (this); } }
class B { void baz (A a) {...} }

//generated remote object for A
class A__remote {
  void  foo  (B  b)  {  b.baz   (this);   }  //"this"   is  of  type  A__remote!
}
```

Method `baz` in class `B` expects an argument of type `A`, hence the call `b.baz(this)` will fail, as `this` is of type `A__remote`. J-Orchestra detects all such explicit uses of `this` and fixes the problem by looking up the corresponding proxy object and replacing `this` with it. Furthermore, we can store the result of the proxy lookup in a local variable and use that variable instead of `this` in future expressions. For example, the rewritten bytecode for `foo` in this case would be:

```
aload_0            //pass "this" to locateProxy method
invokestatic Runtime.locateProxy
checkcast "A"      //locateProxy returns Object, need a cast to "A"
astore_2           //store the located proxy object for future use
```

```
aload_1          //load b
aload_2          //load proxy (of type A)
invokevirtual B.baz
```

At the bytecode level, it is somewhat involved to detect when the transformation should be applied. Recognizing explicit uses of `this` (as opposed to instances of the `aload_0` instruction used to reference the object's own methods) requires a full stack machine emulator for the bytecode instructions. The emulator needs to reconstruct operations and operands from the bytecode stack-machine instruction architecture. This is the only instance where we have found our transformations to be harder to apply at the bytecode level than at the source code level (e.g., like JavaParty does).

4.3.6 Object Identity

To support full object mobility, J-Orchestra assigns globally unique object identifiers to all the remote objects. Each execution site maintains a mapping between remote objects and their proxies. In case of object migration to a remote site, the run-time system first checks whether the site already has a proxy for the remote object. If such a proxy is found, then its remote object field is reassigned. Otherwise, a new proxy object is created. This arrangement preserves correct reference semantics in the presence of full object mobility.

J-Orchestra employs a similar scheme to handle anchored objects' wrapping. When an object is unwrapped and re-wrapped, we should ensure that the identity of the proxy (the "wrap" object) is preserved. This means that the wrapping operation for anchored unmodifiable objects is a bit more complicated than originally presented in Section 4.2.1. Consider an example method `returnMyArgument` in anchored unmodifiable class `A` that takes an argument of another anchored class `B`.

```
B returnMyArgument (B arg) { return arg; }
```

J-Orchestra's rewrite algorithm ensures that the following code fragment preserves its original semantics, although in the translated code all objects will be proxies for application-system translators.

```
B b = new B();
A a = new A();
B b1 = a.returnMyArgument(b);
assert_equal (b == b1);
```

When providing a wrapper for its return value, `returnMyArgument` in the application-system translator for class `A` returns the existing proxy rather than creating a new one.

Another complication results from the fact that Java RMI does not keep a per-site identity for remote objects. If a remotely accessible object is used as a parameter to a remote method, RMI transfers the object's RMI stub. If the stub eventually gets passed back to the site of the original remotely accessible object, the RMI run-time will not recognize that it can use the object directly instead of the stub. Application-system translators need to recognize this case when they are passed a proxy for a locally anchored object, as they need to retrieve a local reference to the anchored object from the proxy. Being able to do this correctly requires maintaining a mapping between application-system translator RMI stubs and the corresponding anchored objects. Fortunately, RMI guarantees the invariant that the identity of a remote object and its stub as provided by the `equals` method is the same. Furthermore, RMI guarantees that the `hashCode` of a remote object and its stub is the same, allowing the mapping to be efficient. An anchored object can be inserted into the mapping using its application-system translator (remote object) and retrieved using the remote object's stub. For those anchored classes that override the `hashCode` and/or `equals` methods providing their own implementations, special care is taken to use the base class (`java.rmi.server.UnicastRemoteObject`) versions of the methods.

4.3.7 Multithreading and Synchronization

The handling of synchronization is an important issue in guaranteeing regular Java semantics for a partitioned multithreaded application. Java has no support for remote synchronization: RMI does not support transparency of synchronization references—all `wait/notify` calls on remote objects are not propagated to the remote site (see [18], section 8.1). Nevertheless, it is possible to build a distributed synchronization mechanism that will guarantee semantics identical to regular Java for all partitioned applications. On the other hand, such a mechanism will likely be complex and inefficient, especially if the distribution relies on an unmodified version of Java RMI. One of the noteworthy issues with synchronization is the possibility of self-deadlocks if thread identity is not maintained when the flow of control moves over the network. We will not describe here the complications of distributed synchronization—a good description of both the problems and the possible solutions (also applicable to J-Orchestra) can be found in the documentation of version 1.05 of JavaParty [8].

In the near future, we plan to evolve the J-Orchestra synchronization mechanism, making this description of transient interest. The current mechanism is rudimentary and incomplete. First, thread identity is not maintained when the flow of control crosses the network, creating the possibility of deadlocks. Second, the identity of locks is guaranteed when `synchronized` *methods* are used (which is the most common Java synchronization technique) but not necessarily when `synchronized` *code blocks* are used. When code blocks are used, lock identity is maintained per-site: if all `synchronized` blocks are executed on the same machine, synchronization will work correctly (barring the problems caused by not maintaining thread identity across machines).

The translation to maintain these properties is as follows: for synchronized methods, we only have to ensure that the proxy "forwarder" method is not synchronized—the original method on the remote object will perform the synchronization. For handling `wait/notify/notifyAll` calls on proxies, we globally detect all such calls and replace them with calls to specially generated methods in the proxy objects (the original `wait/notify/notifyAll` in `java.lang.Object` are `final` and cannot be overridden). Proxies propagate all `wait/notify/notifyAll` calls to the remote objects they represent. All remote objects (`__remote` objects for mobile classes or system/application translators for anchored classes) export methods that implement `wait`/`notify`/`notifyAll` semantics on the object.

4.3.8 Reflection and Dynamic Loading

Reflection can be used explicitly to render the J-Orchestra translation incorrect. For instance, an application class may get an `Object` reference, query it to determine its actual type, and fail if the type is a proxy. Nevertheless, the common case of reflection that is used only to invoke methods of an object is compatible with the J-Orchestra rewrite—the corresponding method will be invoked on the proxy object. In fact, one of the first example applications distributed with J-Orchestra—the JShell command line shell—uses reflection heavily.

We should note that offering full support for correctness under reflection is possible and we have not done so for pure engineering reasons. For example, it is possible to create a J-Orchestra-specific reflection library that will mimic the interface of the regular Java reflection routines but will take care to always hide proxies. All reflection questions on a proxy object will instead be handled by the remote object. With bytecode manipulation, we can replace all method calls to Java reflection functionality with method calls to the J-Orchestra-specific reflection library. We have considered this task to be too complex for the expected benefit.

4.3.9 Similar observations hold regarding dynamic class loading. J-Orchestra is meant for use in cases where the entire application is available and gets analyzed, so that the J-Orchestra classification and translation are guaranteed correct. Currently, dynamically loading code that was not rewritten by J-Orchestra may fail because the code may try to access remote data directly. Additionally, dynamically loading code that calls J-Orchestra rewritten code may violate the security guarantees of the original application (see below). Nevertheless, one can imagine a loader installed by J-Orchestra that takes care of rewriting any dynamically loaded classes before they are used. Essentially, this would implement the entire J-Orchestra translation at load time. Unfortunately, classification cannot be performed incrementally: unmodifiable classes may be loaded and anchored on some nodes before loading another class makes apparent that the previous anchorings are inconsistent. The only safe approach would be to make all dynamically loaded classes anchored on the same network site.

4.3.10 Method Access Modifiers

If methods of a modifiable class are `private` or `protected`, they need to be made `public` if they are to be remotely invokable through Java RMI. J-Orchestra performs this rewrite at the bytecode level. Thus, J-Orchestra does not weaken the compile-time checking of the Java language (the compiler will still check the properties when compiling source code) but it affects the security guarantees of the Java VM. The JavaParty system [13] follows the same approach.

Note that the only potential security problem is with malicious code that calls methods in a J-Orchestra rewritten class. Nevertheless, this is an unusual way to employ J-Orchestra. J-Orchestra is meant for use in cases where the entire application is available and analyzed, so that the J-Orchestra classification and rewriting are guaranteed correct. For code that needs to be dynamically loaded, our previous dynamic loading observations hold: the problem can be solved by a special-purpose, J-Orchestra-aware loader, but we do not offer such a loader yet.

4.3.11 Garbage Collection

Distributed garbage collection is a tough problem. J-Orchestra relies on the RMI distributed reference counting mechanism for garbage collection. This means that cyclic garbage, where the cycle traverses the network, will never be collected. Nevertheless, this aspect is orthogonal to the goal of J-Orchestra—the system just inherits the garbage collection facility of the underlying middleware.

4.3.12 Inner Classes

On the Java language level, inner classes have direct access to all member fields (including private and protected) of their enclosing classes. In order to enable this access, the Java compiler introduces *synthetic* methods that access and modify member fields of enclosing classes. Synthetic methods are not visible during compilation. This clearly presents a problem for J-Orchestra since synthetic methods also need to be accessed through a proxy. The code inside a synthetic proxy method accesses the synthetic method of its remote class. Since proxies are created in source code form, no Java compiler would be able to successfully compile them. Removing the synthetic attributes from methods in remote classes eliminates the problem. The removal does not violate the Java security semantics because there are no access restrictions for synthetic methods to begin with.

4.3.13 System.out, System.in, System.err, System.exit, System.properties

The `java.lang.System` class provides access to standard input, standard output, and error output streams (exported as pre-defined objects), access to externally defined "properties", and a way to terminate the execution of the JVM. In a distributed environment, it is important to modify these facilities so that their behavior makes sense. Different policies may be appropriate for different applications. For example, when any of the partitions writes something to the standard output stream, should the results be visible only on the network site of the partition, all the network sites, or one specially designated network site that handles I/O? If one of the partitions makes a call to `System.exit`, should only the JVM that runs that partition exit or the request should be applied to all the remaining network sites? J-Orchestra allows defining these policies on a per-application basis. For this purpose, J-Orchestra provides classes called `RemoteIn`, `RemoteOut`, `RemoteErr`, `RemoteExit`, and `RemoteProperties` whose implementation determines the application-specific policy. For example, all references to `System.out` are replaced with `RemoteOut.out()` in all the rewritten code. An implementation of `RemoteOut.out()` can return a stream that redirects all the messages to a particular network site, for example.

5 Performance

5.1 Overhead and Limited Rewrite

As mentioned earlier, modifiable classes may be anchored by choice. In fact, it is a common usage scenario for J-Orchestra to try to make mobile only very few classes. We call this the J-Orchestra *limited rewrite* model. The reason to limit which classes are mobile has to do with performance. The J-Orchestra rewrite adds some execution overhead even when mobile objects are used entirely locally. The most significant overheads of the J-Orchestra rewrite are one level of indirection for each method call to a different application object, two levels of indirection for each method call to an anchored system object, and one extra method call for every direct access to another object's fields. The J-Orchestra rewrite keeps overheads as low as possible. For instance, for an application object created and used only locally, the overhead is only one interface call for every virtual call, because proxy objects refer directly to the target object and not through RMI. Interface calls are not expensive in modern JVMs (only about as much as virtual calls [1]) but the overall slowdown can be significant.

The overall impact of the indirection overhead on an application depends on how much work the application's methods perform per method call. A simple experiment puts the costs in perspective. Table 1 shows the overhead of adding an extra interface indirection per virtual method call for a simple benchmark program. The overall overhead rises from 17% (when a method performs 10 multiplications, 10 increment, and 10 test operations) to 35% (when the method only performs 2 of these operations).

Table 1. J-Orchestra indirection overhead as a function of average work per method call (a billion calls total)

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

Penalizing programs that have small methods is against good object-oriented design, however. Furthermore, the above numbers do not include the extra cost of accessing anchored objects and fields of other objects indirectly (although these costs are secondary). To get an idea of the total overhead for an actual application, we measured the slowdown of the J-Orchestra rewrite using J-Orchestra itself as input. That is, we used J-Orchestra to translate the main loop of the J-Orchestra rewriter, consisting of 41 class files totalling 192KB. Thus, the rewritten version of the J-Orchestra rewriter (as well as all system classes it accesses) became remote-capable but still consisted of a single partition. In local execution, the rewritten version was about 37% slower (see Table 2). Although a 37% slowdown of local processing can be acceptable for some applica-