J-Orchestra: Automatic Java Application Partitioning

Eli Tilevich and Yannis Smaragdakis
Center for Experimental Research in Computer Science (CERCS)
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{tilevich, yannis}@cc.gatech.edu

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 is guaranteed to correctly partition any Java program, allowing any application object to be placed on any machine, regardless of how application objects access each other and Java system objects. Additionally, J-Orchestra objects can migrate in response to run-time events in order to take advantage of locality. J-Orchestra also offers run-time optimizations, like the lazy creation of distributed objects—objects do not suffer the overhead of remote registration until they are about to be accessed remotely.

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 more efficiently. Traditional partitioning entails recoding the application functionality so that it uses 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 centralized 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 code.

For several tasks, like switching between local and remote sensor input, automatic partitioning is without direct competition—all other alternatives require that the application be hand-modified. Nevertheless, in the case of user interaction resources (keyboard input, graphical screen output) automatic partitioning finds competition in several existing technologies for transparent distribution. These technologies include Java servlets and text/graphics input/output redirection protocols like telnet and X-Windows [14]. All of the above are rudimentary adaptors for distributed computing: they allow executing a program on a different computer than the one managing the input/output. Nevertheless, all application processing still occurs on a single network site. In contrast, when automatic application partitioning is used, different parts of the application can run on different machines in order to minimize network traffic or reduce server load. For instance, for graphical output, it is often best to keep the code generating the graphics on the same site as the graphics hardware, instead of passing all drawing commands over the network.

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. 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.

Past attempts to automatic partitioning have not scaled to industrial strength applications, for several technical reasons. We argue that J-Orchestra is the most scalable automatic partitioning system in existence. J-Orchestra is the first system that imposes no partitioning constraints on application code: 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. (We will later give precise definitions for the terms “application” and “system” objects, but, roughly, these correspond to instances of regular user classes that do not extend Java system classes, and Java system classes that have native dependencies, respectively.)

To see the scalability advantages of J-Orchestra over prior work, consider the Addistant system [19]—the most mature and closest alternative to J-Orchestra in the design space. J-Orchestra has three advantages over Addistant: a far more general rewrite engine allowing arbitrary partitioning of the application (*generality*); a system supporting object mobility (*flexibility*); and a much lesser dependence on user input for a correct partitioning (*degree of automation*). We examine each of these aspects in turn:

- *Generality*: Addistant imposes restrictions on what applications it can partition and how. For instance, Addistant cannot make a class remotely accessible when the class is unmodifiable and has unmodifiable clients. (The typical reason for a class to be unmodifiable is that its implementation is partly in platform-specific, "native", code, as is the case for many Java system classes.) In general, Addistant decides on a technique for distributing objects on a per-class basis. This means that if even one of the clients of a class needs to access it directly, all clients are restricted to accessing the class directly. Instead, J-Orchestra makes decisions on a per-reference basis. In this way, a single object (e.g., an instance of an unmodifiable Java system class, like `java.io.FileOutputStream`) can be accessed through references of different kinds, depending on the code manipulating each reference. Specifically, the object is accessed directly by other unmodifiable system classes but is accessed through a proxy object by regular (modifiable) application classes. The J-Orchestra rewrite ensures that when a reference is passed from an application class to a system class, it is "unwrapped", to produce a direct reference so that the system code can access the object directly. Similarly, when a reference is passed from system code to application code, the object is "wrapped": a reference to a proxy is generated, so that the application code accessing the object can migrate anywhere on the network. This mechanism is responsible for the generality of J-Orchestra: any application object can be on any machine, regardless of what other objects it references.

- *Flexibility*: Addistant does not allow object mobility. Objects are created and are used on the same network site. In contrast, J-Orchestra application objects can freely migrate to different network sites at run-time—e.g., to take advantage of locality. This makes J-Orchestra a much more flexible system: migration policies can be put in place and get activated in response to run-time events, instead of fixing object locations once and for all. For instance, a method call may cause the arguments of the method to migrate to the site where the method code is executed.

- *Degree of Automation*: Addistant requires user input for every application and system class. The user input determines the semantics of remote object access. For instance, the Addistant user has to explicitly specify whether instances of an unmodifiable class are created only by modifiable code, whether an unmodifiable class is accessed by modifiable code, whether instances of a class can be safely passed by-copy, etc. (As indicated above, the list is not exhaustive: there are cases that the Addistant arsenal of rewrite techniques does not cover.) In contrast, J-Orchestra does not require the user to know how classes are implemented and what their referencing behavior is. This elevates the degree of automation in the system—the user performing the partitioning no longer needs to have a sophisticated understanding of the application semantics.

  Furthermore, J-Orchestra offers the ability to rewrite a limited portion of the application to make it remotely accessible. In this way, not all application classes need to be accessible through proxy objects—the application can operate as it normally would in a centralized environment, except for the parts that need to be accessible remotely. This is a desirable property because proxy indirection may slow down application execution by a factor of some tens of percent. By only rewriting a small portion of the application, we ensure high-speed local execution without sacrificing remote accessibility. Addistant can provide a similar benefit, but in J-Orchestra this feature is usable automatically with guaranteed correctness. J-Orchestra provides static analysis tools that automatically determine the unmodifiable classes that can potentially be used by an application. This information can then be used to determine the minimal rewriting actions that need to be performed to render any subset of the application remotely accessible.

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 the optimizations that we have implemented in J-Orchestra and present performance measurements that demonstrate the advantage of J-Orchestra over input/output redirection with X-Windows.

2 REWRITE STRATEGY OVERVIEW

In this section, we give a high-level overview of the J-Orchestra rewrite algorithm. In our discussion, we assume that all objects in the application are to be turned into remotely accessible objects. This assumption simplifies our argument of the generality of J-Orchestra. In Section 4.1, we will discuss how the assumption can be safely relaxed.

2.1 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. The invariant maintained is that clients never get direct references to objects that can potentially be remote—access is always through a proxy. Application code needs to be rewritten to maintain this invariant: for instance, all `new` statements have to be rewritten to create a proxy object and return it, an object has to be prevented from passing direct references to itself (as the value of the `this` expression) 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, like 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 *unmodifiable code*. Typically, code is unmodifiable because it is in a platform-specific or “native” form—the implementation of 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.

2.2 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 environment.

- If the client code (i.e., user 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., user 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., user 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 the remote object can be referred to directly by unmodifiable clients and indirectly by modifiable clients. 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 should 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 to 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. Thus, when a method of such a proxy is called, the reference arguments need to be *unwrapped* before the method 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.

  Essentially, instead of the usual call-by-value semantics of Java method calls, our proxies implement a *call-by-value-convert* semantics, where the references passed as arguments are automatically converted exactly when (and if) needed.

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. The reason is that such objects cannot be referenced directly by unmodifiable code. (An exception is the case of application classes that extend system classes other than `java.lang.Object`—we will discuss such complications in our detailed presentation of the J-Orchestra rewrite model.) In contrast, instances of Java system classes are remotely accessible but typically cannot migrate, as they may be accessed directly by other system objects.

3 REWRITE MECHANISM

In this section, we discuss in concrete detail the J-Orchestra rewrite model. Several elements that were previously elided are presented thoroughly.

We will first give precise definitions of our terminology in order to classify the different types of classes J-Orchestra deals with.

J-Orchestra converts all objects of an application into *remote-capable* objects. Remote-capable objects can be accessed from a remote site. We distinguish three kinds of remote-capable object classes: *mobile* classes, *anchored unmodifiable* classes, and *anchored modifiable* classes. The “anchored/mobile” attribute refers to run-time behavior. *Anchored* classes can be accessed remotely but cannot move through the network. *Mobile* classes can migrate at will.

We should emphasize that the mechanisms of *classification* and *translation* of classes are entirely separate. J-Orchestra uses a conservative algorithm to determine whether an object should be anchored or mobile. This algorithm could change in the future, affecting the way classes are categorized. Nevertheless, the translation mechanism for mobile classes, anchored unmodifiable classes, and anchored modifiable classes can stay 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 may write “class `A` refers to class `B`” to mean that an instance of `A` may hold a reference to some instance of `B`.

3.1 Classification

For simplicity in our classification, we assume that the application to be partitioned is written in pure Java (i.e., the only access to native code is inside Java system classes). This is the standard scenario where J-Orchestra is used. Our observations can be straightforwardly generalized to applications that include some native code.[1]

In principle, classes need to be anchored when they provide abstractions for machine-specific services and resources such as threading (`java.lang.Thread`) or I/O (`java.io.ObjectOutputStream`). Other classes may need to be anchored because of the way they interact with anchored classes. For instance, if an anchored class directly accesses the fields of another object, that object should also be anchored. In fact, such accesses may not be apparent to the Java code as they may occur in native code. Therefore, we often need to be conservative and assume that native code can potentially directly reference the fields of all parameters passed to it.

The J-Orchestra classification algorithm for the vast majority of classes can be summarized as follows. (Some exceptions will be discussed individually.)

Anchored Unmodifiable (System) Classes. A system class `C` is *anchored unmodifiable* if it depends on native code (i.e., has native methods), or references to `C` objects can be passed between application code and an anchored unmodifiable class.

Anchored Modifiable (Application) Classes. A class is *anchored modifiable* if it is a modifiable application class that extends an anchored unmodifiable class (other than `java.lang.Object`).

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.

The interesting distinction in the above classification is between system classes that are mobile and system classes that are anchored. Note that even classes that do *not* reference native code and are *not* referenced by native code may need to be anchored, as long as their instances are passed to/from anchored system classes. 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.

Java system classes are mobile, if 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.[2] 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.

As an advanced technical note, we should mention that less conservative 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. Specifically, 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 in all applicable cases—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 anchored system classes in the Swing system library—for instance the `javax.swing.table.DefaultTableModel` 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 (and which) 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. The only exception has to do with arrays and will be discussed in Section 3.3.4.

1. If the application includes native code, our guarantees will need to be similarly 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.

2. It is not clear whether this replication is allowed under the legal conditions of JDK usage. In the long run, if replication turns out to be impossible, an inheritance approach is feasible, but requires more engineering work (because of the lack of multiple inheritance).

3.2 Translation

3.2.1 Anchored System Classes

System classes are anchored in groups: an anchored system class needs to be co-located with all related anchored system classes. Unrelated anchored classes, however, can be located on different machines. In practice, 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 the Java Development Kit (JDK) supports hierarchical organization through the concept of packages. A Java package contains classes that share common functionality. JDK classes within the same package reference mostly each other and very rarely instances of the 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.

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.[3] 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 greater 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 `anchored.java.lang.Thread`. To make remote execution possible, all application 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 application 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`. 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 wrapping, 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));
```

3. The existence of a separate application-system translator is an implementation detail—under different middleware, the translator functionality could be folded inside the proxy. J-Orchestra currently uses Java RMI as its distribution middleware. 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.

```
}

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 remote application-system translator.

3.2.2 Anchored Application Classes

Anchored application classes are the application classes that inherit from anchored system classes. Recall that anchored system classes depend on platform-specific resources and thus cannot migrate through the network. Anchored application classes can be thought of as referencing native libraries indirectly through their superclasses. Anchored application classes are handled with a translation that is identical to the one for anchored system classes, except for one aspect. The defining distinction between system and application anchored classes is that the latter can access other application classes' fields directly. Such direct field accesses have to be detected and replaced with accessor and mutator methods. In this way, other application classes referenced by anchored application classes do not need to be anchored.

Consider an arbitrary method `foo` of an anchored application class `MyThread` that extends `java.lang.Thread`.

```
class MyThread extends java.lang.Thread {
  void foo (A a) { a.counter++; }
}
```

This clearly creates a problem. If class `A` is placed on a remote network site, field `counter` can no longer be accessed directly since RMI (and all other distribution frameworks) are method-based. To fix the problem, J-Orchestra replaces direct accesses with accessor and mutator methods and adds those methods to class `A`.

```
class MyThread extends java.lang.Thread {
  void foo (A a) {
    a.set$$counter (a.get$$counter () + 1);
  }
}
```

3.2.3 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. The clients of a mobile class access its proxy in exactly the same way as they used to access the original class.

A remote class is responsible for handling the network execution semantics. Remote classes 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`. Since it is the proxies that inherit the names of the original classes, remote classes must be consistently renamed. J-Orchestra gives remote classes an "`__remote`" suffix. 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
class A extends B implements I, Proxy { ... }

//Remote class declaration
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.

3.3 Handling of Java Language Features

The J-Orchestra rewrite has to handle several Java language features. Some parts of the translation (e.g., that of static methods) are straightforward and only add engineering complexity. Handling some 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). In other cases, however, J-Orchestra has a higher obligation than JavaParty to maintain local execution semantics for a partitioned application, since J-Orchestra partitioning is automatic for the entire application.

3.3.1 Static Methods and Fields

J-Orchestra has to handle remote execution of static methods. This also takes care of remote access to static fields: J-Orchestra rewrites all direct accesses to fields (both member and static) of other classes with 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.

3.3.2 *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.

3.3.3 *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) { ... }
}
```

3.3.4 *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 application 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.

3.3.5 *"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 now 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
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 occurrences of the `aload_0` instruction that are used to reference to the object's own methods) requires a stack machine emulator for the bytecode instructions. The emulator needs to reconstruct operations and operands from the bytecode stack-machine instruction architecture.

3.3.6 *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 proxies and their remote objects. In case of remote object migration, the run-time system first checks whether the remote object already has a proxy on the current host. 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 objects is a bit more complicated than originally presented in Section 3.2.1. Consider an example method `returnMyArgument` in anchored 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. Being able to do this correctly requires maintaining a mapping between application-system translators and their corresponding anchored objects.

This mapping also helps solve the problem of Java RMI not keeping a per-site identity for its 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. Stated differently, identity is not preserved for remotely accessible objects passed to remote methods. This complicates the unwrapping operation performed by application-system translators. It would be impossible to retrieve the corresponding anchored object from an application-system translator stub without some additional information. Fortunately, RMI guarantees the invariant that the `hashCode` method returns the same value whether invoked on a remote object or its stub. This property makes keeping the aforementioned mapping between anchored objects and their application-system translators possible. 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` method providing their own implementation, special care is taken to use the base class (`java.rmi.server.UnicastRemoteObject`) version of the method.

3.3.7 Multithreading and Synchronization

The handling of synchronization is an important issue in guaranteeing regular Java semantics for a partitioned multithreaded application. Java 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). We are currently in the process of implementing a full synchronization system for J-Orchestra. This system will guarantee semantics identical to regular Java for all partitioned applications. Our system is similar to the mechanism for transparent synchronization used in version 1.05 of JavaParty (see [8]). We believe, however, that we can address the JavaParty problems with blocks synchronized on remote objects. (We also believe that our solution could be implemented for JavaParty, as well.) Nevertheless, since this solution is not fully implemented, we do not describe it here.

The currently implemented J-Orchestra synchronization approach guarantees correctness 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, J-Orchestra guarantees correct synchronization per-site: if all `synchronized` blocks are executed on the same machine, synchronization will work correctly.

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.

3.3.8 Inner Classes

Inner classes were added to JDK1.1 without introducing changes to the JVM instruction set. At the bytecode level, inner classes are supported by `synthetic` methods and inner class attributes. Synthetic methods in a class can only be used by its inner classes. Since it might make sense to place an inner class and its enclosing class on different network sites, J-Orchestra completely eliminates all the inner class dependencies from its remote-capable classes. This means removing inner class and synthetic method attributes along with consistently renaming all the inner classes. For example, all the references to `Outer$Inner` will be replaced with `Outer_Inner`. This streamlines the compilation process for the generated proxies of inner classes. Removing all the inner classes dependencies allows placing each proxy in a separate file and compiling proxies in an arbitrary order.

3.3.9 Handling System.out, System.in, System.err, System.exit, System.properties

The `java.lang.System` class provides access to several system facilities exported by the JVM. Among these facilities are standard input, standard output, and error output streams (exported as predefined objects), access to externally defined "properties", and a way to terminate the execution of the JVM. All these facilities assume having a single JVM and are not aware of distribution. In a distributed environment, it is important to modify the aforementioned 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.

4 PERFORMANCE

4.1 Optimizations

4.1.1 Limited Rewrite

Up to this point, we have discussed a J-Orchestra translation where *every* application and system class is made remote-capable. This simplifies the presentation of the J-Orchestra translation mechanism. Nevertheless, in practice, we mostly use J-Orchestra with a rewrite technique that affects as few classes as possible. We call this the J-Orchestra *limited rewrite* model.

The reason to limit which classes get rewritten has to do with performance. The full J-Orchestra rewrite adds some execution overhead to the application even when objects are used entirely locally. Specifically, the J-Orchestra rewrite adds 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. These overheads are kept 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 (they cost approximately 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 suffices to put 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 re-written 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 re-written version was about 37% slower (see Table 2). Although a 37% slowdown of local processing can be acceptable for some applications, for many others it is too high.

For this reason, J-Orchestra offers the ability to rewrite a limited portion of an application to make it remotely accessible. In this way, the application can operate with no overhead, as it normally would in a centralized environment, except for the parts that need to be accessible remotely. Given a set of classes that must be remotely accessible, the J-Orchestra's static analysis tools automatically determine what other application classes must be remotely accessible as well. For example, the user might indicate that all the UI classes must be remotely accessible. This information can then be used to determine the minimal rewriting actions that need to be performed to render the UI subset of the application remotely accessible. If certain classes need to be mobile, all the non-mobile classes they reference have to be remotely accessible. The limited rewrite process is fully automated.

In general, limited rewrite can be viewed as a version of the J-Orchestra full rewrite, where many of the application classes are explicitly not made mobile, but just anchored or even not affected at all (if they are only accessed by anchored classes, they do not need to be remote-capable). Conceptually, these classes are now on the system side of the application/system boundary, and, thus, they use direct references to all other anchored objects.

The limited rewrite is particularly successful when most of the processing in an application occurs on one network site and only some resources (e.g., graphics, sound, keyboard input) are accessed remotely. We have used the limited rewrite to partition several applications that follow this pattern (e.g., a GUI-driven demo of the Java speech API, a graphical display of real time statistics from another machine, etc.). In all cases, the execution overhead from J-Orchestra indirection was practically zero.

4.1.2 Lazy Remote Object Creation

Recall that remote objects extend `java.rmi.server.UnicastRemoteObject` to enable remote execution. The constructor of `java.rmi.server.UnicastRemoteObject` exports the remote object to the RMI run-time. This is an intensive process that significantly slows down the overall object creation. J-Orchestra tries to avoid this slowdown by employing lazy remote object creation for all the objects that might never be invoked remotely. If a proxy constructor determines that the object it wraps is to be created on the local machine, then the creation process does not go through the object factory. Instead, a *lazy* version of the remote object is created directly. A lazy object is identical to a remote one with the exception of having a different name and not inheriting from `java.rmi.server.UnicastRemoteObject`. A proxy continues to point to such a lazy object until the application attempts to use that proxy in a remote method call. In that case, the proxy converts its lazy object to a remote one using a special conversion constructor. This constructor reassigns every member field from the lazy object to the remote one. All static fields are kept in the remote version of the object to avoid data inconsistencies.

Although this optimization may at first seem RMI-specific, in fact it is not. Every middleware mechanism suffers significant overhead for registering remotely accessible objects. Lazy remote object creation ensures that the overhead is not suffered until it is absolutely necessary. In the case of RMI, our experiments show that the creation of a remotely accessible object is over 200 times more expensive than a single constructor invocation! In contrast, the extra cost of converting a lazy object into a remotely accessible one is about the same as a few variable assignments in Java. Therefore, it makes sense to optimistically assume that objects are created only for local use, until they are actually passed to a remote site. Considering that a well-partitioned application will only move few objects over the network, the optimization is likely to be valuable.

The impact of speeding up object creation is significant in terms of total application execution time. We measured the effects using the J-Orchestra code itself as a benchmark. The result is shown below. The measurements are on the full J-Orchestra rewrite: all objects are made remote-capable, although they are executed on a single machine. 767 objects were constructed during this execution. The overhead for the version of J-Orchestra that eagerly constructs all objects to be remote-capable is 58%, while the same overhead when the objects are created for local use is less than 38% (an overall speedup of 1.15, or 15%).

Table 2: Effect of lazy remote object creation and J-Orchestra indirection on total execution time

Original time	Indirect lazy	Overhead	Indirect non-lazy	Overhead
6.63s	9.11s	37.4%	10.48s	58.1%

4.2 Performance Measurements

J-Orchestra is an attractive alternative to input/output redirection technologies like X-Windows and telnet. In this section, we compare the performance of J-Orchestra to X-Windows, used to display graphics on a remote host.

The trade-off in all our experiments is simple: X-Windows has a lower overhead per network transfer, but J-Orchestra has tremendous flexibility to place the drawing code on the machine where the graphics will be displayed. More specifically, if J-Orchestra and X-Windows perform the same number of network operations, X-Windows will always be faster. This may come as a surprise since the X protocol [15] for transferring graphics over the network is commonly considered a “heavy” protocol. Nevertheless, compared to a heavyweight implementation of general purpose middleware like Java RMI, the X protocol is fairly lightweight. A major difference, for instance, is that most X protocol requests do not generate replies, but RMI remote method calls will always need to generate network traffic when an operation completes. On the other hand, J-Orchestra can outperform X-Windows, because J-Orchestra can altogether avoid transferring the drawing commands over the network and instead issue them locally.

All the experiments described are partitioned using simple manual input, consisting of location constraints for only a couple of classes in the application. The J-Orchestra limited rewrite then detects which other application and system classes need to be rewritten and whether they should be anchored or mobile. In all experiments, we measured the run time of the original Java application, as well as the run time of the rewritten (i.e., remote-capable) version of the application but executing in a single partition. These two baseline results were identical—the limited rewrite only adds indirection to a tiny proportion of the total objects created in our example programs.

We used JDK 1.3 on two Sun Ultra 10 machines (Sparc II 440MHz processor) connected with a 100Mbit Ethernet network for these experiments.

4.2.1 Window Drawing

We created three different tests of window operations. The first opens an empty remote window. The second opens a remote window and displays 100 text buttons on it. The third opens a remote window and displays 100 graphical buttons on it. In all three cases, the window is repainted 10 times. Each of the three experiments has two versions: one where all drawing operations are initiated from the window object itself and one where the (re-)painting is initiated from a different object. The reason for this last distinction is that we want to produce a more “realistic” comparison by initiating the operations remotely. That is, in the J-Orchestra case, there will be operations over the network for each re-painting, although the graphics for the buttons themselves will never need to be transferred over the network.

The results (run times) are shown below (all numbers are averages of 3 runs that varied by at most 0.5s). The baseline is the run time of a local version.

Table 3: Version 1 of window experiments

Experiment/ System	Empty window	Window + 100 text buttons	Window + 100 graphics buttons
Baseline	2.9s	7.2s	6.6s
X-Windows	4.7s	8.2s	15.8s
J-Orchestra	3.1s	7.7s	6.6s

Table 4: Version 2 of window experiments

Experiment/ System	Empty window	Window + 100 text buttons	Window + 100 graphics buttons
Baseline	2.7s	7.6s	6.8s
X-Windows	4.5s	8.5s	16.3s
J-Orchestra	4.9s	8.4s	7.7s

Version 1 of the above experiment shows the benefit of J-Orchestra, but the partitioning can be considered “unfairly optimal”. All the graphics are produced in response to a single network operation. Therefore, J-Orchestra performs very close to the baseline in the Version 1 experiment. Version 2, however, is more realistic: all re-drawing is initiated over the network. In this case, J-Orchestra performs about the same as X-Windows, except for the case of graphics buttons. In this case, X-Windows has to transfer the graphical icon over the network, while J-Orchestra avoids this overhead altogether. As a result, J-Orchestra is more than twice as fast as X-Windows. Of course, a slower network (e.g., 10Mbit ethernet, ISDN, or modem connection) would accentuate these results dramatically. We should also mention that the window with text buttons does not display correctly in the case of X-Windows (an empty window is displayed).

4.2.2 Simple Animation

In this benchmark, we test a small but fully usable third-party application. This experiment is representative of the way X-Windows and J-Orchestra will be used in practice. It consists of a Java analog clock program (one of the many written as Java graphics demos). The program draws a simple face of an analog clock (60 minute marks, 4 hour numbers, and three moving clock hands). With X-Windows, we just run the clock application on one machine and display the results on another. With J-Orchestra, however, we can transfer only the interesting data (a current `Date` object) over the network and do all the drawing locally. To turn this into a useful benchmark, we changed it very slightly, so that the clock updates the time on the screen as quickly as possible—i.e., the program keeps polling the system for time as often as it can and displays the results on screen. The measured quantity is then the frames-per-second attained on the remote display. In other words, we are treating the clock display as a real-time animation and measure the animation quality. This is representative of actual uses of J-Orchestra. J-Orchestra is useful when the user needs to display or process real-time results on a different machine than the one producing the results.

The measurements (frames per second) for this benchmark appear below. Apart from the original clock, we also created two stripped-down versions. The first only draws the moving parts of the clock (i.e., the hands). The second draws the clock hands as well as the numbers “3”, “6”, “9”, and “12” on the face of the clock.

Table 5: Clock Experiment

Experiment/ System	Original clock	Clock with just hands	Clock with hands and hours
Baseline	56 fps	104 fps	77 fps
X-Windows	20 fps	74 fps	33 fps
J-Orchestra	42 fps	68 fps	61 fps

For the original clock application, J-Orchestra is more than twice as fast as X-Windows. The reason is that J-Orchestra only needs to transfer the time information. In contrast, X-Windows needs to redraw the minute mark lines and the numbers “3”, “6”, “9”, and “12” on the face of the clock, even though these do not change in time. The corresponding drawing commands have to be issued over the network, which slows down X-Windows execution significantly.

Actually, J-Orchestra is much faster than X-Windows for the clock application, even though it has to overcome two disadvantages. First, because of the way the application is written, J-Orchestra has to transfer more data than strictly needed—a complete `java.util.Date` object needs to be passed over the network, and this contains more information than just the current time (e.g., year, date, and time zone). Second, the underlying protocol (Java RMI) is less efficient than X-Windows in transferring the required data. On the other hand, J-Orchestra also has an advantage over X-Windows. Although J-Orchestra transfers more data, it transfers all the data at once in a single `Date` object, thus incurring network latency only once. X-Windows has to issue a separate drawing operation over the network for each line or text string displayed.

The stripped-down version of the clock demonstrates the effect of these two factors. As shown in Table 5, if just the clock hands were drawn, J-Orchestra would have been a little slower than X-Windows. If, however, as little as the four hour numbers (3, 6, 9, and 12) need to be drawn on the face of the clock, J-Orchestra again is much faster than X-Windows.

5 RELATED WORK

Distributed computing has been the main focus of systems research in the past two decades. Therefore, there is a wealth of work that exhibits similar goals or methodologies to ours. We will separate closely related work (approaches that use similar techniques to ours) from indirectly related work (work with similar goals but significantly different approaches).

5.1 Directly Related Work

Several recent systems other than J-Orchestra can also be classified as automatic partitioning tools. In the Java world, the closest approaches are the Addistant [19] and Pangaea [16] systems. The Coign system [9] has promoted the idea of automatic partitioning for applications based on COM components. We discussed Addistant extensively in Section 1, so we will concentrate on the other two systems here.

Coign [9] is an automatic partitioning system for software based on Microsoft’s COM model. Although Coign is a pioneering system, it suffers from two drawbacks. First, Coign is not applicable to many real-world situations: although Windows software often exports coarse-grained COM components, very few real-world applications are written as collections of many fine-grained COM components. The applications that constitute success cases for Coign (mainly the Octarine word processor) were experimental and written specifically to showcase that COM is a viable platform for developing applications from many small components. The second drawback is technical. Coign does not try to solve the hard problems of automatic partitioning: it does not distribute components when they share data through memory pointers. Such components are deemed non-distributable and are located on the same machine. Practical experience with Coign [9] showed that this is a severe limitation for the only real-world application included in Coign’s example set (the Microsoft PhotoDraw program). The Coign approach would be impossible in the case of Java: almost all program data are accessed through references in Java. No support for synchronous data mobility exists in Coign, but the application can be periodically repartitioned based on its recent behavior.

Pangaea [16][17] is an automatic partitioning system that has very similar goals to J-Orchestra. Pangaea is based on the JavaParty [13] infrastructure for application partitioning. Since JavaParty is designed for manual partitioning and operates at the source code level, Pangaea is also limited in this respect. Thus, Pangaea cannot be used to make Java system classes (which are supplied in bytecode format) remotely accessible. Therefore, Pangaea has little applicability to real world situations, especially with limited manual intervention. For instance, much data exchange in Java programs happens through system classes (e.g., collection classes, like `java.util.Vector`). If such classes are not remotely accessible, all their clients need to be located on the same site, making partitioning almost impossible for realistic applications.

Finally, we should mention that the JavaParty infrastructure [13][8] is closely related to J-Orchestra. The similarity is not so much in the objectives—JavaParty only aims to support manual partitioning and does not deal with system classes. The techniques used, however, are very similar to J-Orchestra, especially for the newest versions of JavaParty [8].

5.2 Indirectly Related Work

Automatic partitioning is essentially a *Distributed Shared Memory (DSM)* technique. Just like traditional DSM approaches, we try to create the illusion of a shared address space, when the data are really distributed across different machines. Nevertheless, automatic partitioning differs from traditional DSM work in one major aspect: *only the application is allowed to change, not the run-time environment*. Traditional DSM systems like Munin [5], Orca [3], and, in the Java world, CJVM [2], and Java/DSM [22] use a specialized run-time environment in order to detect access to remote data and ensure data consistency. The deployment cost of DSMs has restricted DSM applicability to high-performance parallel applications. In contrast, automatically partitioned Java applications work on original, unmodified Java Virtual Machines (JVMs), possibly shipped with Web browsers. All modifications necessary are made directly to the application, using compilation techniques. In this way, automatic partitioning has no deployment cost, allowing it to be applied to regular applications and compete with lightweight technologies like X-Windows.

Among distributed shared memory systems, the ones most closely resembling the J-Orchestra approach are object-based DSMs, like Orca [3]. The Orca system has a dedicated language and run-time system, but also has similarities to J-Orchestra in its treatment of data at the object level, and its use of static analysis.

Mobile object systems, like Emerald [4][10] have similarities with J-Orchestra. Many of the J-Orchestra ideas on implementing mobile objects and choosing appropriate semantics for method invocations (synchronous object migration) have originated with Emerald.

The Doorastha system [6] represents another piece of work closely related to automatic partitioning. Doorastha allows the user to annotate a centralized program to turn it into a distributed application. Unfortunately, all the burden is shifted to the user to specify what semantics are valid for a specific class (e.g., whether objects are mobile, whether they can be passed by-copy, etc.). The Doorastha annotations are quite expressive in terms of how method arguments, different fields of a class, etc., are manipulated. Nevertheless, programming in this way is tedious and error-prone: a slight error in an annotation may cause insidious inconsistency errors.

The need for infrastructure to support application partitioning has been recognized in the systems community. Proposals for such infrastructure (most recently, Protium [21]) usually try to address different concerns from those covered by J-Orchestra. High performance is an essential element, with the infrastructure trying to hide the latency of remote accesses. J-Orchestra aims at a much higher degree of automation, but for applications with more modest network performance requirements.

Finally, we should mention that the overall approach of programming distributed systems as if they were centralized ("papering over the network") has been occasionally criticized (e.g., see the best known "manifesto" on the topic [20]). The main point of criticism has been that distributed systems fundamentally differ from centralized systems because of the possibility of partial failure, which needs to be handled differently for each application. Nevertheless, J-Orchestra can address this problem, at least partially: although the input of the system is a binary application, the proxies for remote-capable classes are produced in source code. Application-specific partial-failure handling can be effected by manually editing the source code of the proxy classes and handling the corresponding Java language exceptions. Thus, although J-Orchestra hides much of the complexity of distribution, it allows the user to handle distribution-specific failure exactly like it would be handled through manual partitioning. Alternatively viewed, the user can concentrate on the part of the application that really matters for distributed computing: partial failure handling. This part is the only code that needs to be written by hand in order to partition an application.

6 STATUS AND CONCLUSIONS

J-Orchestra is work-in-progress, but most of the back-end functionality is in place, as described in this paper. We have already used J-Orchestra to partition several realistic, third-party applications. Among them are "J-Shell" (a command line shell implementation for Java), a graphical demo of the Java speech API (the user selects parameters and a sound synthesizer composes phrases), an application for monitoring server load and displaying real-time graphical statistics, and some small graphical demos and benchmarks. All of the above were partitioned in a client-server model, where the I/O part of the functionality (graphics, text, etc.) is displayed on a client machine, while processing or execution of commands takes place on a server. Our client machine is typically a hand-held iPAQ PDA, running Linux. This environment is good for showcasing the capabilities of J-Orchestra—even relatively uninteresting centralized applications become exciting demos when they are automatically turned into distributed applications, partly running on a hand-held device that communicates over a wireless network with a central server.

In the future, we intend to continue work on the J-Orchestra backend, but at the same time place more emphasis on front-end functionality. The existing J-Orchestra GUI is limited: it does not allow the specification of any mobility properties and does not interface well with the rewrite functionality. Most of the J-Orchestra rewrites are currently triggered programmatically (using scripts). An integrated environment is necessary to improve the system's third-party usability. A lot more work is also required on the distributed performance aspects. Currently, J-Orchestra uses Java RMI as its distribution middleware. RMI has been criticized for its inefficiency, but offers useful features for transparent distribution (e.g., distributed garbage collection). In the future, we may select a more efficient middleware implementation (e.g., KaRMI [12]) when such alternatives become more mature. Any middleware, however, will perform badly if the application is not partitioned well and object mobility is not coordinated optimally. Therefore, the greatest future challenge for J-Orchestra will be to develop mechanisms that automatically infer detailed object migration strategies in response to synchronous events. (For example, a strategy could be as detailed as “when a method `foo` is called, all its arguments and all data reachable from its arguments in up to three indirections should migrate to the method’s execution site.”)

A common question we are asked concerns our choice of the name “J-Orchestra”. The reason for the name is that there is a strong analogy between application partitioning and the way orchestral music is often composed. Many orchestral pieces are not originally written for orchestral performance. Instead, only a piano score is originally composed. Later, an “orchestration” process takes place that determines which instruments should play which notes of the completed piano score. There are many examples of orchestrating piano music that was never intended by its composer for orchestral performance. There are several examples of piano pieces that have several brilliant but totally different orchestrations. With J-Orchestra, we provide a state-of-the-art “orchestration” facility for Java programs. Taking into account the unique capabilities of network nodes (instruments) we partition Java applications for harmonious distributed execution. We believe that automatic application partitioning represents a huge promise and that J-Orchestra is the first general and scalable automatic partitioning tool.

ACKNOWLEDGMENTS

Austin (Chun Fai) Chau, Dean Pu Mao, Kane See, and Hailemelekot Seifu have all contributed to the J-Orchestra front-end (GUI and profiler) tools as well as the partitioning and set up of current J-Orchestra demo applications. Their enthusiasm helped us stay on track.

This work has been supported by the Yamacraw Foundation, by DARPA/ITO under the PCES program, and by a Raytheon E-Systems faculty fellowship.

REFERENCES

[1] Bowen Alpern, Anthony Cocchi, Stephen Fink, David Grove, and Derek Lieber, “Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless”, in Proc. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[2] Yariv Aridor, Michael Factor, and Avi Teperman, “CJVM: a Single System Image of a JVM on a Cluster”, in Proc. *ICPP’99*.

[3] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Ruhl, and M. Frans Kaashoek, “Performance Evaluation of the Orca Shared-Object System”, *ACM Trans. on Computer Systems*, 16(1):1-40, February 1998.

[4] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, “Distribution and Abstract Types in Emerald”, in *IEEE Trans. Softw. Eng.*, 13(1):65-76, 1987.

[5] John B. Carter, John K. Bennett, and Willy Zwaenepoel, “Implementation and performance of Munin”, *Proc. 13th ACM Symposium on Operating Systems Principles*, pp. 152-164, October 1991.

[6] Markus Dahm, “Doorastha—a step towards distribution transparency”, *JIT*, 2000. See `http://www.inf.fu-berlin.de/~dahm/doorastha/`.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification, 2nd Ed.*, The Java Series, Addison-Wesley, 2000.

[8] Bernhard Haumacher, Jürgen Reuter, Michael Philippsen, “JavaParty: A distributed companion to Java”, `http://wwwipd.ira.uka.de/JavaParty/`

[9] Galen C. Hunt, and Michael L. Scott, “The Coign Automatic Distributed Partitioning System”, *3rd Symposium on Operating System Design and Implementation (OSDI’99)*, pp. 187-200, New Orleans, 1999.

[10] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black, “Fine-Grained Mobility in the Emerald System”, ACM Trans. on Computer Systems, 6(1):109-133, February 1988.

[11] Nelson King, “Partitioning Applications”, *DBMS and Internet Systems* magazine, May 1997. See `http://www.dbmsmag.com/9705d13.html`.

[12] Christian Nester, Michael Phillipsen, and Bernhard Haumacher, “A More Efficient RMI for Java”, in Proc. *ACM Java Grande Conference*, 1999.

[13] Michael Philippsen and Matthias Zenger, “JavaParty - Transparent Remote Objects in Java”, *Concurrency: Practice and Experience*, 9(11):1125-1242, 1997.

[14] Robert W. Scheifler, and Jim Gettys, “The X Window System”, *ACM Transactions on Graphics*, 5(2): 79-109, April 1986.

[15] Robert W. Scheifler, “X Window System Protocol, Version 11”, *Network Working Group RFC 1013*, April 1987.

[16] Andre Spiegel, “Pangaea: An Automatic Distribution Front-End for Java”, 4th *IEEE Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '99)*, San Juan, Puerto Rico, April 1999.

[17] Andre Spiegel, “Automatic Distribution in Pangaea”, *CBS 2000*, Berlin, April 2000. See also `http://www.inf.fu-berlin.de/~spiegel/pangaea/`

[18] Sun Microsystems, Remote Method Invocation Specification, `http://java.sun.com/products/jdk/rmi/`, 1997.

[19] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano, “A Bytecode Translator for Distributed Execution of ‘Legacy’ Java Software”, *European Conference on Object-Oriented Programming (ECOOP)*, Budapest, June 2001.

[20] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall, “A note on distributed computing”, Technical Report, Sun Microsystems Laboratories, SMLI TR-94-29, November 1994.

[21] Cliff Young, Y. N. Lakshman, Tom Szymanski, John Reppy, David Presotto, Rob Pike, Girija Narlikar, Sape Mullender, and Eric Grosse, “Protium, and Infrastructure for Partitioned Applications”, *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. May 20—23, 2001, Schoss Elmau Germany, pp. 41-46, IEEE Computer Society Press, 2001.

[22] Weimin Yu, and Alan Cox, “Java/DSM: A Platform for Heterogeneous Computing”, *Concurrency: Practice and Experience*, 9(11):1213-1224, 1997.