# CHAPTER IV

# J-ORCHESTRA

This chapter presents J-Orchestra, an automatic partitioning system for Java programs. J-Orchestra takes as input a Java program in bytecode format and transforms it into a distributed application, running across multiple Java Virtual Machines (JVMs). To accomplish such automatic partitioning, J-Orchestra substitutes method calls with remote method calls, direct object references with proxy references, and so forth, by means of bytecode rewriting and code generation. The partitioning does not involve any explicit programming or modifications to the JVM or its standard runtime classes. The main novelty and source of scalability of J-Orchestra is its approach to dealing with unmodifiable code (e.g., Java system classes). The approach consists of an analysis heuristic that determines which application objects get passed to which parts of native (i.e., platform-specific) code and a technique for injecting code that will convert objects to the right representation so that they can be accessed correctly inside both application and native code. Validating the type information accuracy and testing the correctness of the analysis heuristic have demonstrated its viability in the J-Orchestra context. To be able to run partitioned programs over a standard remote procedure call middleware such as RMI, J-Orchestra introduces a new approach to maintaining the Java centralized concurrency and synchronization semantics over RMI efficiently. Finally, specialized domains present opportunities for making J-Orchestra partitioning more automatic, which is the case for appletizing—a semi-automatic

approach to transforming a Java GUI application into a client-server application, in which the client runs as a Java applet that communicates with the server through RMI.

## 4.1 Introduction

Adding distributed capabilities to existing programs has come to the forefront of software evolution [44] and is commonly accomplished through application partitioning—the task of splitting up the functionality of a centralized monolithic application into distinct entities running across different network sites. As a programming activity, application partitioning entails re-coding parts of the original application so that they could interact with a distributed middleware mechanism such as Remote Procedure Call (RPC) [10] or Common Object Request Broker Architecture (CORBA) [61]. In general, this manual process is costly, tedious, error prone, and sometimes infeasible due to the unavailability of source code, as in the case of many commercial applications.

Automating, even partially, a tedious and error-prone software development task is always a desirable goal. Thus, automating application partitioning would not only save programming time but would also result in an effective approach to separating distribution concerns. Having a tool that under human guidance handles all the tedious details of distribution could relieve the programmer of the necessity to deal with middleware directly and to understand all the potentially complex data sharing through pointers.

Automating any programming task presents an inherent dichotomy between power and automation: any automation effort hinders complete control for users with advanced requirements. Indeed, transforming a centralized application for distributed execution often requires changes in the logic and structure of the application to satisfy such requirements

as fault tolerance, load balancing, and caching. In view of this dichotomy, one important question is what kind of common architectural characteristics make applications amenable to automatic partitioning, and when meaningful partitioning is impossible without manually changing the structure and logic of the application first.

J-Orchestra operates on binary (Java bytecode) applications and enables the user to determine object placement and mobility to obtain a meaningful partitioning. The application is then re-written to be partitioned automatically and different parts can run on different machines, on unmodified versions of the Java VM. For a large subset of Java, the resulting partitioned application's execution semantics is identical to the one of its original, centralized version. The requirement that the VM not be modified is important. Specifically, changing the runtime is undesirable both because of deployment reasons (it is easy to run a partitioned application on a standard VM) and because of complexity reasons (Java code is platform-independent, but the runtime system has a platform-specific, native-code implementation).

The conceptual difficulty of performing application partitioning in general-purpose languages (such as Java, C#, but also C, C++, etc.) is that programs are written to assume a shared memory: an operation may change data and expect the change to be visible through all other pointers (*aliases*) to the same data. The conceptual novelty of J-Orchestra (compared to past partitioning systems [33][75][84] and distributed shared memory systems [2][3][5][14][102]) consists of addressing the problems resulting from inability to analyze and modify all the code under the control flow of the application. Such unmodifiable code is usually part of the runtime system on which the application is running. In the case of Java, this runtime is the Java VM. In the case of free-standing applications, the runt-

ime is the OS. Without complete control of the code, execution is in danger of letting a reference to a remote object get to code that is unaware of remoteness. Prior partitioning systems have ignored the issues arising from unmodifiable code and have had limited scalability, as a result. J-Orchestra features a novel rewrite mechanism that ensures that, at runtime, references are always in the expected form ("direct" = local or "indirect" = possibly remote) for the code that handles them. The result is that J-Orchestra can split code that deals with system resources, safely running, e.g., all sound synthesis code on one machine, while leaving all unrelated graphics code on another.

This chapter starts by describing the general partitioning approach of J-Orchestra and its analysis algorithm and rewriting engine. Then it covers how J-Orchestra maintains the Java centralized concurrency and synchronization semantics over RMI efficiently. Finally, it demonstrates how specialized domains present opportunities to make J-Orchestra partitioning more automatic through the case of appletizing.

Chapter V of this dissertation identifies the environment features that make J-Orchestra possible and argues that partitioning systems following the principles laid out by J-Orchestra are valuable in modern high-level run-time systems such as the Java VM or Microsoft's CLR. Chapter V also presents several case-studies that demonstrate J-Orchestra handling arbitrary partitioning of realistic applications without requiring an understanding of their internals.

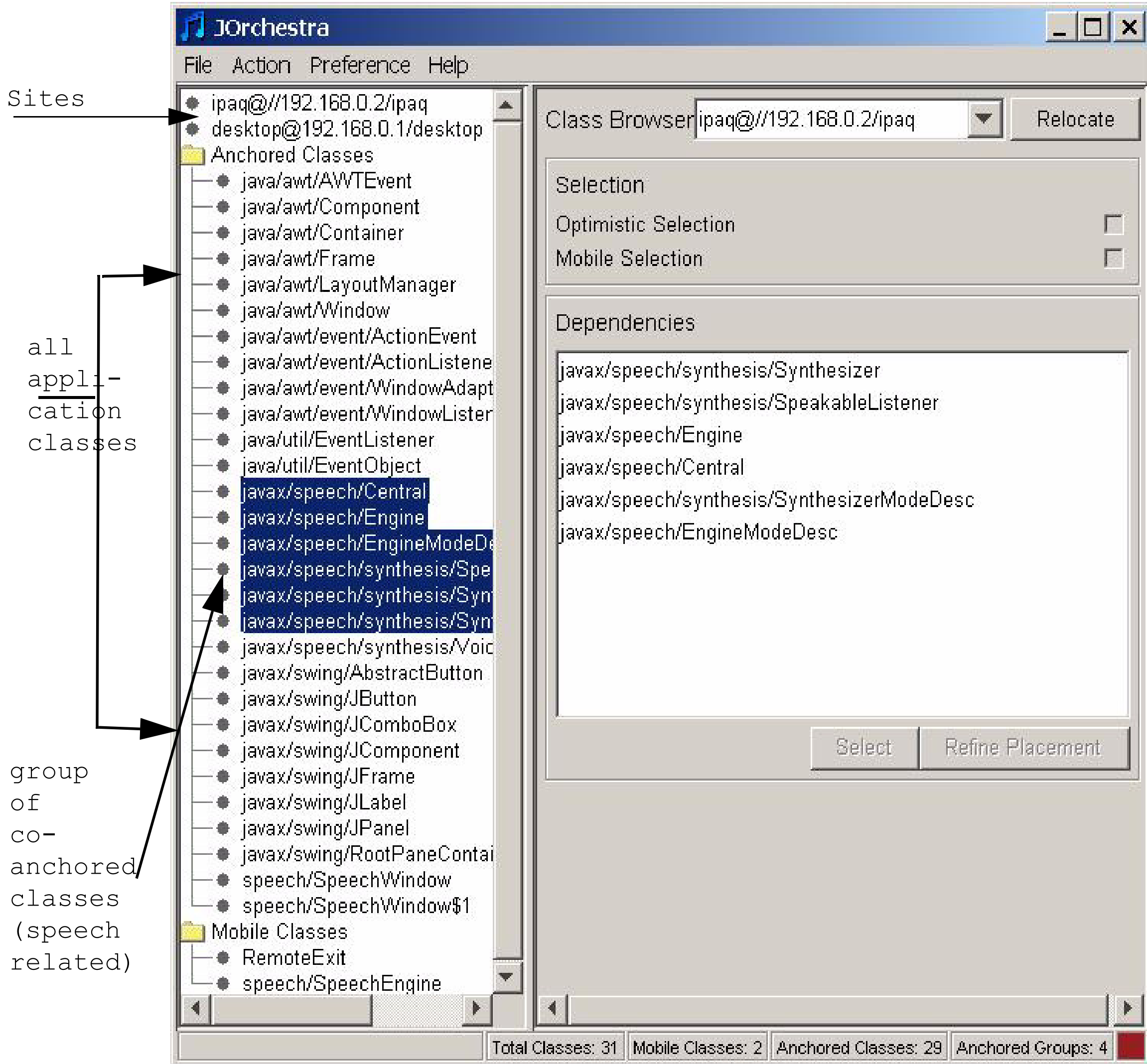


**Figure 4-1:** Example user interaction with J-Orchestra. An application controlling speech output is partitioned so that the machine doing the speech synthesis is different from the machine controlling the application through a GUI.

## 4.2 User View of J-Orchestra

Figure 4-1 shows a screenshot of J-Orchestra in the process of partitioning a small but realistic example application. The original example Swing application showcases the Java Speech API and works as follows: the user chooses predefined phrases from a drop-down box and the speech synthesizer pronounces them. As a motivation for partitioning,

imagine a scenario in which this application needs to be run on a small device such as a PDA that either has no speakers (hardware resource) or does not have the Speech API installed (software resource). The idea is to partition the original application in a client-server mode so that the graphical partition (i.e., the GUI), running on a PDA, would control the speech partition, running on a desktop machine. We chose this particular example because it fits well into the realm of applications amenable for automatic application partitioning. The locality patterns here are very clear and defined by the specific hardware resources (graphical screen and speech synthesizer) and their corresponding classes (Swing and Speech API).

Figure 4-1 shows J-Orchestra at a point when it has finished importing all the referenced classes of the original application and has run its classification algorithm (Section 4.4) effectively dividing them into two major groups represented by tree folders *anchored* and *mobile*.

- Anchored classes control specific hardware resources and make sense within the context of a single JVM. Their instances must run on the JVM that is installed on the machine that has the physical resources controlled by the classes. J-Orchestra clusters anchored classes into groups for safety; intuitively, classes within the same anchored group reference each other directly and as such must be co-located during the execution of the partitioned application. If classes from the same group are placed on the same machine, the partitioned application will never try to access a remote object as if it were local, which would cause a fatal run-time error. J-Orchestra classification algorithm (Section 4.4) has created four anchored groups for this example. One group contains all the referenced speech API classes. The remaining groups specify various Swing classes. While classes within the same anchored group cannot be separated, anchored groups can be placed on different network sites. In our example, all the Swing classes anchored groups should be

placed on the site that will handle the GUI of the partitioned application to obtain meaningful partitioning.

- Mobile classes do not reference system resources directly and as such can be created on any JVM. Mobile classes do not get clustered into groups, except as an optimization suggestion. Instances of mobile classes can move to different JVMs independently during the execution to exploit locality. Supporting mobility requires adding some extra code to mobile classes at translation time to enable them to interact with the runtime system. Mobility support mechanisms create overhead that can be detrimental for performance if no mobility scenarios are meaningful for a given application. To eliminate this mobility overhead, a mobile class can be *anchored by choice*. We discuss anchoring by choice and its implications on the rewriting algorithm in Section 4.5.2.

The J-Orchestra GUI represents each network node in the distributed application by a dedicated tree folder. The user then drag-and-drops classes from the anchored and mobile folders to their destination network site folder. Putting an anchored class in a particular network folder assigns its final location. For a mobile class, it merely assigns its initial creation location. Later, an instance of a mobile object can move as described by a given mobility policy. When all classes are assigned to destination folders, the J-Orchestra rewriting tool transforms the original centralized application into a distributed application. At the end, J-Orchestra puts all the modified classes, generated supporting classes, and J-Orchestra runtime configuration files into `jar` files, one per destination network site.

At run-time, J-Orchestra employs its runtime service to handle such tasks as remote object creation, object mobility, and various bookkeeping tasks.

## 4.3 The General Problem and Approach

In abstract terms, the problem that J-Orchestra solves is *emulating a shared memory abstraction for unaware applications without changing the runtime system*. The following two observations distinguish this problem from that of related research work. First, the requirement of not changing the run-time system while supporting unaware applications sets J-Orchestra apart from traditional Distributed Shared Memory (DSM) systems. (The related work chapter (Chapter VII) offers a more complete comparison.) Second, the implicit assumption is that of a pointer-based language. It is conceptually trivial to support a shared memory abstraction in a language environment in which no sharing of data through pointers (aliases) is possible. Although it may seem obvious that realistic systems will be based on data sharing through pointers,[1] the lack of data sharing has been a fundamental assumption for some past work in partitioning systems—e.g., the Coign approach [33].

It is worth asking why mature partitioning systems have not been implemented in the past. For example, why no existing technology allows the user to partition a platform-specific binary (e.g., an x86 executable) so that different parts of the code can run on different machines? We argue that the problem can be addressed much better in the context of a high-level, object-oriented runtime system, like the JVM or the CLR, than in the case of a platform-specific binary and runtime. The following three concrete problems need to be overcome before partitioning is possible:

1. The pointers may be hidden from the end user (e.g., data sharing may only take place inside a Haskell monad). The problems identified and addressed by J-Orchestra remain the same regardless of whether the end programmer is aware of the data sharing or not.

1. The granularity of partitioning has to be coarse enough: the user needs to have a good vocabulary for specifying different partitions. High-level, object-oriented runtime systems, like the Java VM, help in this respect because they allow the user to specify the partitioning at the level of objects or classes, as opposed to memory words.

2. It is necessary to establish a mechanism that adds an indirection to every pointer access. This involves some engineering complexity, especially under the requirement that the runtime system remain unmodified.

3. The indirection has to be maintained even in the presence of unmodifiable code. Unmodifiable code is usually code in the application's runtime system. For example, in the case of a stand-alone executable running on an unmodified operating system, the program may create entities of type "file" and pass them to the operating system. If these files are remote, a runtime error will occur when they are passed to the unsuspecting OS. Addressing the problem of adding indirection in the presence of unmodifiable code is the main novelty of J-Orchestra. This problem, in different forms, has plagued not just past partitioning systems but also traditional Distributed Shared Memory systems. Even page-based DSMs often see their execution fail because protected pages get passed to code (e.g., an OS system call expecting a buffer) that is unaware of the mechanism used to hide remoteness.

We now look at the problem in more detail, in order to see the complications of adding indirection to all pointer references. The standard approach to such indirection is to convert all direct references to indirect references by adding proxies. This creates an abstraction of shared memory in which proxies hide the actual location of objects—the actual object may be on a different network site than the proxy used to access it. This abstraction is necessary for correct execution of the program across different machines because of *aliasing*: the same data may be accessible through different names (e.g., two different pointers) on different network sites. Changes introduced through one name/pointer

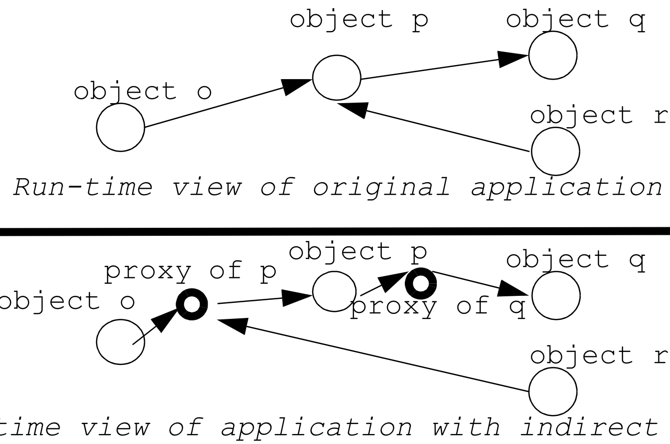


**Figure 4-2:** Results of the indirect reference approach schematically. Proxy objects could point to their targets either locally or over the network.

should be visible to the other, as if on a single machine. Figure 4-2 shows schematically the effects of the indirect referencing approach. This indirect referencing approach has been used in several prior systems [66][74][84].

Since one of our requirements is to leave the runtime system unchanged, we cannot change the JVM's pointer/reference abstraction. Instead, J-Orchestra rewrites the entire partitioned application to introduce proxies for every reference in the application. Thus, when the original application would create a new object, the partitioned application will also create a proxy and return it; whenever an object in the original application would access another object's fields, the corresponding object in the partitioned application would have to call a method in the proxy to get/set the field data; whenever a method would be called on an object, the same method now needs to be called on the object's proxy; etc.

The difficulty of this rewrite approach is that it needs to be applied to *all code that might hold references to remote objects*. In other words, this includes not just the code of the original application but also the code inside the runtime system. In the case of the Java

VM, such code is encapsulated by system classes that control various system resources through native code. Java VM code can, for instance, have a reference to a thread, window, file, etc., object created by the application. However, not being able to modify the runtime system code, one can not make it aware of the indirection. For instance, one cannot change the code that performs a file operation to make it access the file object correctly for both local and remote files: the file operation code is part of the Java VM (i.e., in machine-specific binary code) and partly implemented in the operating system. If a proxy is passed instead of the expected object to runtime system code that is unaware of the distribution, a run-time error will occur. Without changing the platform-specific runtime (JVM+OS) of the application, one cannot enable remoteness for all of the code.[2] (For simplicity, the implicit assumption is that the application itself does not contain native code—i.e., it is a "pure Java" application.)

J-Orchestra effectively solves many of the problems of dealing with unmodifiable code by partitioning *around* unmodifiable code. This approach consists of the following two parts. The first is the classification algorithm: a static analysis that determines which classes should be co-located. The second is the rewrite algorithm, which inserts the right code in the partitioned application so that, at run-time, indirect references are converted to direct and vice versa when they pass from mobile to anchored code. In order to perform classification, even though one cannot analyze the platform-specific binary code for every platform, J-Orchestra employs a heuristic that relies on the type information of the inter-

2. It is interesting to compare the requirements of adding indirection to those of a garbage collector. A garbage collector needs to be aware of references to objects, even if these references are manipulated entirely through native code in a runtime system. Additionally, in the case of a copying collector, the GC needs to be able to change references handled by native code. Nonetheless, being aware of references and being able to change them is not sufficient in our case: we need full control of all code that manipulates references, since the references may be to objects on a different machine and no direct access may be possible.

faces to the Java runtime (i.e., the type signatures of Java system classes). This is another way in which high-level, object-oriented runtime systems make application partitioning possible.

The end result is that, unlike past systems [66][74][84], J-Orchestra ensures safety while imposing a minimum amount of restrictions on the placement of objects in a pure Java application. Under the assumption that the classification heuristic is correct, which it typically is, the programmer does not need to worry about whether remote objects can ever get passed to unmodifiable code. Additionally, objects can refer to system objects through an indirection from everywhere on the network. If they need to ever pass such references to code that expects direct access, a direct reference will be produced at run-time.

Next we describe the three major technical components of J-Orchestra—the classification heuristic, the translation engine, and the handling of concurrency and synchronization.

## 4.4 Classification Heuristic

The J-Orchestra classification algorithm [87] classifies each class as anchored or mobile and determines anchored class groups. Classes in an anchored group must be placed on the same network site since they access each other directly.

The purpose of the classification algorithm is to determine the rewriting strategy that J-Orchestra must follow to enable the indirect referencing approach for each class in the partitioned application. In other words, classification informs the rewriter about generating and injecting code, as opposed to having the user specify this information manually. We have already described the first criterion of classification: each class can be either anchored

| MOBILITY \ MODIFIABILITY | Y | N |
|---|---|---|
| Y | Application<br>System | |
| N | Application<br>System | System |

**Figure 4-3:** J-Orchestra classification criteria. For simplicity, we assume a “pure Java” application: no unmodifiable application classes exist.

or mobile. The second criterion deals with modifiability properties of a class: each class is either modifiable or not. A class is unmodifiable if its instances are manipulated by native code (e.g., if it has native methods or if its instances may be passed to native methods of other objects). Such dependencies inhibit the spectrum of changes one can make to the class’s bytecode (sometimes none) without rendering it invalid. Figure 4-3 presents a diagram that shows all possible combinations of the classification criteria. As the diagram depicts, J-Orchestra distinguishes between three categories of classes: *mobile*, *anchored modifiable*, and *anchored unmodifiable*.

By examining the J-Orchestra classification criteria in Figure 4-3, one can draw several observations about the relationship between mobility and modifiability. One is that only a modifiable class can be rewritten so that its instances could participate in object mobility scenarios. I.e., unmodifiable mobile quadrant does not have any entries. Another observation is that only systems classes can be unmodifiable (in a “pure Java” application), and all unmodifiable systems classes are anchored. Finally, both application and systems classes can be mobile and modifiable.

Before presenting the rules that J-Orchestra follows to classify a class as unmodifiable, we demonstrate the idea informally through examples. Consider class `java.awt.Component`. This class is anchored unmodifiable because it has a native method `initIDs`. It is anchored because it must remain on the site of native platform specific runtime libraries on which it depends. It is unmodifiable because modifying its bytecode could render it invalid. As an example of a destructive modification, consider changing the class's name. Because the class's name is part of a key that matches native method calls in the bytecode to their actual native binary implementations, the class would no longer be able to call its native methods. A more general reason for not modifying the bytecode of an unmodifiable class is that because native code may be accessing directly the object layout (e.g., reading object fields). Having native methods, however, is not the only condition that could make it possible for instances of a class to be passed to native code. Consider class `java.awt.Point`, which does not have any native dependencies. However, `java.awt.Component` has a method `contains` that takes a parameter of type `java.awt.Point`. Because `java.awt.Component` is unmodifiable, its `contains` method can take only an instance of the original class `java.awt.Point` rather than its proxy—method `contains` could be accessing the fields of its `java.awt.Point` parameter directly. Therefore, if `java.awt.Point` is used in the same program (along with `java.awt.Component`), its classification category would be anchored as well. Furthermore, in the J-Orchestra methodology, we refer to such classes as *co-anchored*, meaning that because of the possibility of accessing each other directly, these classes must be kept together on the same site throughout the execution.

Conceptually, the classification heuristic has a simple task. It computes for each class $A$ and $B$ an answer to the question: *can references to objects of class A leak to unmodifiable (native) code of class B*? If the answer is affirmative, $A$ cannot be remote to $B$: otherwise the unmodifiable code will try to access objects of class $A$ directly (e.g., to read their fields), without being aware that it accesses an indirection (i.e., a proxy) resulting in a run-time error. This criterion determines whether $A$ and $B$ both belong to the same anchored group. If no constraint of this kind makes class $A$ be part of an anchored group, and class $A$ itself does not have native code, then it can be mobile. Next we present a heuristic, consisting of four basic rules, through which J-Orchestra co-anchors classes to anchored groups. Each co-anchored group must stay on the same site throughout the distributed execution. These rules essentially express a transitive closure, and the J-Orchestra classification iterates them until it reaches a fixed point.

1. Anchor a system class with native methods.
2. Co-anchor an anchored class with system classes used as parameters or return types of its methods or static methods.
3. Co-anchor an anchored class with the system class types of all its fields or static fields.
4. Co-anchor a system class, other than `java.lang.Object`, with its subclasses and superclasses.

The following few points are worth emphasizing about our classification heuristic:

- The above rules represent the essence of the analysis rather than its exhaustive description. The abbreviated form of the rules improves readability, especially since the analysis is based on heuristic assumptions, and therefore we do not make an argument of strict correctness.

- Specifically, the rules do not mention arrays or exceptions—these are handled similarly to regular classes holding references to the array element type and method return types, respectively. In addition, an array type is considered together with all its constituent types (e.g., an array type `T[][]` has constituent types `T[]` and `T`).

- Not all access to application objects inside native code/anchored classes is prohibited—only access that would break if a proxy were passed instead of the real object. Notably, the rules ignore Java interfaces: interface access from unmodifiable code is safe and imposes no restriction. Indeed, anchored unmodifiable code can even refer to mobile objects and to anchored objects in different groups through interfaces. The reason is that an interface does not allow direct access to an object: it does not create a name dependency to a specific class, and it cannot be used to access object fields. Because a proxy can serve just as well as the original object for access through an interface, distribution remains transparent for interface accesses.

- The rules codify a simple type-based heuristic. It computes all types that get passed to anchored code, based on information in the type signatures of methods and the calling information in the methods (in either application or system classes) that consist of regular Java bytecode. This is a conservative approach, as it only provides analysis on a per-type granularity and always assumes the worst: if an instance of class `A` can be passed to native code, all instances of any subtype of `A` are considered anchored (we make an explicit exception for `java.lang.Object`, or no partitioning would be possible).

- Despite the conservatism, however, the algorithm is not safe! The unsafety is inherent in the domain: no analysis algorithm, however conservative or sophisticated, can be safe if the unmodifiable code itself cannot be analyzed. The real data flow question we would like to ask is "what objects get accessed directly (i.e., in a way that would break if a proxy were used) by unmodifiable code?" The fully conservative answer is "all objects" since unmodifiable code can perform arbitrary side-effects and is not realistically analyzable, because it is only available in platform-specific binary form. Thus, unmodifiable code in the Java VM could (in theory) be accessing directly *any* object created by the application. For example, when an application creates a `java.awt.Component`, it is possible that some other, seemingly unrelated native system code, will maintain a ref-

erence to this object and later access its fields directly, preventing that code from running on a remote machine.

- In the face of inherent unsafety, our classification is an engineering approximation. We rely on the rich type interfaces and on the informal conventions used to code the Java system services. Specifically, we make the following three engineering assumptions about native code behavior. First, we assume that classes without native methods do not have specialized semantics (i.e., no object is accessed by unmodifiable code unless it is passed at some point in the program explicitly to such code through a native method call). This assumption also implies that all system objects are created under direct application control rather than spontaneously in the native code. Second, we assume that system classes's type information is strong, and that the system services do not discover type information not present in the type signatures (i.e., native code does not make assumptions about an `Object` reference passed to it by dynamically discovering the real type of the object and accessing its fields directly). Finally, we assume that native code does not share state between different pieces of functionality such as I/O, graphics, and sound (i.e., native code controlling different system resources are autonomous entities that can be safely separated to run on different JVMs).

- Although the assumptions of our classification heuristic are arbitrary, it is important to emphasize again that any different assumptions would be just as arbitrary: safety is impossible to ensure unless either partitioning is disallowed (i.e., a single partition is produced) or platform-specific native code can be analyzed. Since the classification analysis will be heuristic anyway, its success or failure is determined purely by its scalability in practice. We present empirical evidence on the accuracy of the first two assumptions in Chapter VI, and our experience of partitioning multiple applications that use different sets of native resources has confirmed the last assumption to be well-founded as well. As we discuss in Section 4.9, because our assumptions do not hold in certain cases (e.g., for `Thread` objects, or implicit objects like `System.in`, `System.out`), we provide specialized treatment for such objects.

- A more exact (less conservative) classification algorithm would be possible. For example, we could perform a data-flow analysis to determine which objects can leak to

unmodifiable code on a per-instance basis. The current classification heuristic, however, fits well the J-Orchestra model of type-based partitioning (recall that the system is semi-automatic and does not assume source code access: the user influences the partitioning by choosing anchorings on a per-class basis). Choosing a more sophisticated algorithm is orthogonal to other aspects of J-Orchestra. In particular, the J-Orchestra rewriting engine (Section 4.5) will remain valid regardless of the analysis used. In practice, J-Orchestra allows its user to override the classification results and explicitly steer the rewrite algorithm.

Our discussion so far covered modifiable and anchored unmodifiable classes, but left out *anchored modifiable classes*. The vast majority of these classes are not put in this category by the classification algorithm. Instead, these classes could be mobile, but are anchored by choice by the user of J-Orchestra. As briefly mentioned earlier, anchoring by choice is useful because it lets the class's code access all co-anchored objects without suffering any indirection penalty. Some of the anchored modifiable classes, however, are automatically classified as such by the classification heuristic. These classes are direct subclasses of anchored unmodifiable classes with which they are co-anchored. An application class `MyComponent` that extends `java.awt.Component` would be an example of such a class. This class does not have any native dependencies of its own, but it inherits those dependencies from its super class. As a result, both classes have to be co-anchored on the same site. Since `MyComponent` is an application class, it can support some limited bytecode manipulations. For example, it is possible to change bytecodes of individual methods or add new methods without invalidating the class. At the same time, changing `MyComponent`'s superclass would violate its intended original semantics. That is why J-Orchestra must follow a different approach to enable remote access to anchored modifiable classes.

## 4.5 Rewriting Engine

Having introduced and evaluated the J-Orchestra classification heuristic, we can now describe how the classification information gets used. The J-Orchestra rewriting engine is parameterized with the classification information. The classification category of a class determines the set of transformations it goes through during rewriting. The term "rewriting engine" is a slight misnomer due to the fact that applying binary changes to existing classes is not the only piece of functionality required to enable indirect referencing. In addition to bytecode manipulation,[3] the rewriting engine generates several supporting classes and interfaces in source code form. Subsequently, all the generated classes get compiled into bytecode using a regular Java compiler. We next describe the main ideas of the rewriting approach.

### 4.5.1 General Approach

The J-Orchestra rewrite first makes sure that all data exchange among potentially remote objects is done through method calls. That is, every time an object reference is used to access fields of a different object and that object is either mobile or in a different anchored group, the corresponding instructions are replaced with a method invocation that will get/set the required data.

For each mobile class, J-Orchestra generates a proxy that assumes the original name of the class. A proxy class has the same method interface as the original class and dynamically delegates to an implementation class. Implementation classes, which get generated by binary-modifying the original class, come in two varieties: *remote* and *local-only*. The

3. We use the BCEL library [18] for bytecode engineering.

difference between the two is that the remote version extends `UnicastRemoteObject` while the local-only does not. Subclasses of `UnicastRemoteObject` can be registered as RMI remote objects, which means that they get passed by-reference over the network. I.e., when used as arguments to a remote call, RMI remote objects do not get copied. A remote reference is created instead and can be used to call methods of the remote object.

Local-only classes are an optimization that allows those clients that are co-located on the same JVM with a given mobile object to access it without the overhead of remote registration. (We discuss the local-only optimization in Section 4.8.1—for now it can be safely ignored.) The implementation classes implement a generated interface that defines all the methods of the original class and extends `java.rmi.Remote`. Remote execution is accomplished by generating an RMI stub for the remote implementation class. We show below a simplified version of the code generated for a class.

```
//Original mobile class A
class A {
 void foo () { ... }
}

//Proxy for A (generated in source code form)
class A implements java.io.Externalizable {
 //ref at different points can point either to
 //local-only or remote implementations, or RMI stub.
 A__interface ref;
 ...
 void foo () {
  try {
   ref.foo ();
  } catch (RemoteException e) {
    //let the user provide custom failure handling
  }
 }//foo
}//A
```

```
//Interface for A (generated in source code form)
interface A__interface extends java.rmi.Remote {
 void foo () throws RemoteException;
}

//Remote implementation (generated in bytecode form
//by modifying original class A)
class A__remote extends UnicastRemoteObject implements
       A__interface {
 void foo () throws RemoteException {...}
}

//Local-only version is identical to remote
//but does not extend UnicastRemoteObject
```

Proxy classes handle several important tasks. One such task is the management of globally unique identifiers. J-Orchestra maintains an "at most one proxy per site" invariant via the help of such globally unique identifiers. Each proxy maintains a unique identifier that it uses to interact with the J-Orchestra runtime system. All proxies implement `java.io.Externalizable` to take full control of their own serialization. This enables the support for object mobility: at serialization time proxies can move their implementation objects as specified by a given mobility scenario. Note that proxy classes are generated in source code, thus enabling the sophisticated user to supply handling code for remote errors.

For anchored classes, proxies provide similar functionality but do not assume the names of their original classes. Since both modifiable and unmodifiable anchored classes cannot change their superclass (to `UnicastRemoteObject`), a different mechanism is required to enable remote execution. An extra level of indirection is added through special purpose classes called *translators*. Translators implement remote interfaces and their purpose is to make anchored classes look like mobile classes as far as the rest of the J-Orchestra rewrite is concerned. Regular proxies, as well as remote and local-only implementation versions are created for translators, exactly like for mobile classes. The code generator puts

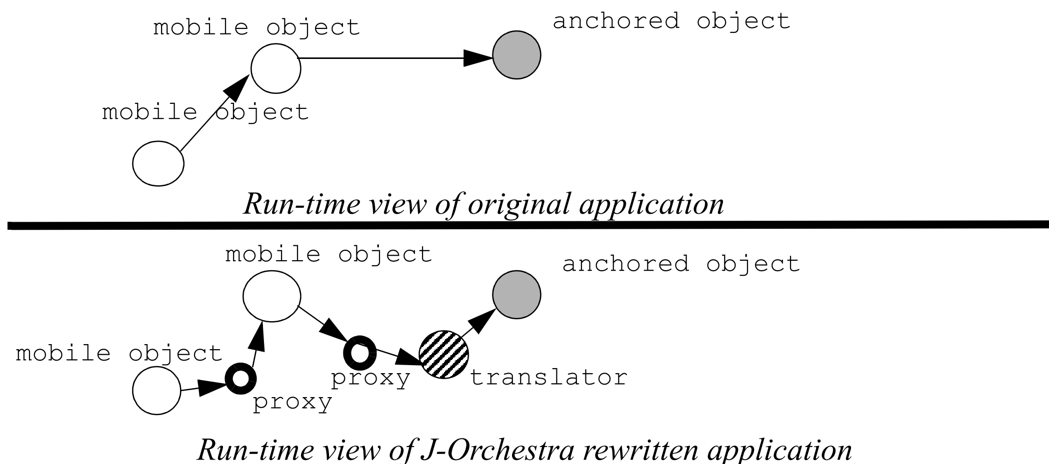


**Figure 4-4:** Results of the J-Orchestra rewrite schematically. Proxy objects could point to their targets either locally or over the network.

anchored proxies, interfaces and translators into a special package starting with the prefix `remotecapable`. Since it is impossible to add classes to system packages, this approach works uniformly for all anchored classes. Figure 4-4 shows schematically what an object graph looks like during execution of both the original and the J-Orchestra rewritten code. The two levels of indirection introduced by J-Orchestra for anchored classes can be seen. Note that proxies may also refer to their targets indirectly (through RMI stubs) if these targets are on a remote machine.

In addition to giving anchored classes a "remote" identity, translators perform one of the most important functions of the J-Orchestra rewrite: the dynamic translation of direct references into indirect (through proxy) and vice versa, as these references get passed between anchored and mobile code. Consider what happens when references to anchored objects are passed from mobile code (or anchored modifiable code as we will see in the next section) to anchored code. For instance, in Figure 4-5, a mobile application object `o` holds a reference `p` to an object of type `java.awt.Point`. Object `o` can pass reference `p` as an

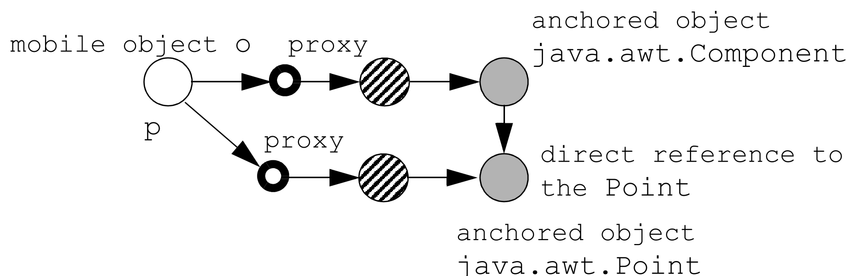


**Figure 4-5:** Mobile code refers to anchored objects indirectly (through proxies) but anchored code refers to the same objects directly. Each kind of reference should be derivable from the other.

argument to the method `contains` of a `java.awt.Component` object. The problem is that the reference `p` in mobile code is really a reference *to a proxy* for the `java.awt.Point` but the `contains` method cannot be rewritten and, thus, expects a direct reference to a `java.awt.Point` (for instance, so it can assign it or compare it with a different reference). In general, the two kinds of references should be implicitly convertible to each other at run-time, depending on what kind is expected by the code currently being run.

It is worth noting that past systems that follow a similar rewrite as J-Orchestra [31][66][74][76][84] do not offer a translation mechanism. Thus, the partitioned application is safe only if objects passed to unmodifiable (system) code are guaranteed to always be on the same site as that code. This is a big burden to put on the user, especially without analysis tools, like the J-Orchestra classification tool. With the J-Orchestra classification and translation, no object will ever be accessed directly if it can possibly be remote. (See Section 4.9 for some limitations.)

Translation takes place when a method is called on an anchored object. The translator implementation of the method "unwraps" all method parameters (i.e., converts them

from indirect to direct) and “wraps” all results (i.e., converts them from direct to indirect). Since all data exchange between mobile code and anchored code happens through method calls (which go through a translator) we can be certain that references are always of the correct kind. For a code example, consider invoking (from a mobile object) methods `foo` and `bar` in an anchored class `C` passing it a parameter of type `P`. Classes `C` and `P` are packaged in packages `a` and `b`, respectively, and are co-anchored on the same site. The original class `C` and its generated translator are shown below (slightly simplified):

```
//original anchored class C
package a;
class C {
 void foo (b.P p) {...}
 b.P bar () { return new b.P(); }
}

//translator for class C
package remotecapable.a;
class C__translator extends UnicastRemoteObject implements
               C__interface {
 a.C originalC;
 ...
 void foo (remotecapable.b.P p) throws RemoteException {
  originalC.foo ((b.P) Runtime.unwrap(p));
 }

 remotecapable.b.P bar() throws RemoteException {
  return (remotecapable.b.P)Runtime.wrap(originalC.bar());
 }
}
```

#### 4.5.2 Call-Site Wrapping for Anchored Modifiable Code

In the previous section we presented the dynamic conversion of references when calls are made to methods of anchored objects by mobile objects. Nevertheless, wrapping and unwrapping need to also take place when (modifiable) anchored (usually by-choice) objects call other anchored objects that are in a different anchored group. This case is more

complex, but handling it is valuable as it is the only way to enable anchoring by choice. This section explains in detail the wrapping mechanism for anchored modifiable objects.

Anchored and mobile classes present an interesting dichotomy. Anchored objects call methods of all of their co-anchored objects directly without any overhead. Accesses from anchored objects to anchored objects of a different anchored group, on the other hand, result in significant overhead (see Section 4.8) for every method call (because of proxy and translator indirection) and field reference (because direct field references are rewritten to go through method calls). Mobile objects suffer a slightly lower overhead for indirection: calling a method of a mobile object, irrespective of the location of the caller, always results in a single indirection overhead (for the proxy). At the same time, mobile objects can move at will to exploit locality. The result is that if objects of a modifiable class tend to be accessed mostly locally and only rarely remotely, it can be advantageous to anchor this class by choice. In this way, no indirection overhead is incurred for accesses to methods and fields of co-anchored objects. An anchored modifiable class is still remotely accessible (like all classes in a J-Orchestra-rewritten application) but proxies are only used for true remote access.

From a practical standpoint, anchoring by choice is invaluable. It usually allows an application to execute with no slowdown, except for calls that are truly remote. Anchoring by choice 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.

Translators of anchored classes, as discussed in the previous section, are the only avenue for data exchange between mobile objects and anchored objects. Translators are a

simple way to perform the wrapping/unwrapping operation because there is no need to analyze and modify the bytecode of the caller: the call is just indirected to go through the translator, which always performs the necessary translations. This approach is sufficient, as long as all the control flow (i.e., the method calls) happens *from* the outside *to* the anchored group but an anchored object never calls methods of objects outside its group. This is the case for pure Java applications consisting of only mobile and anchored unmodifiable (i.e., system) objects. In this case, system code is unaware of application objects and can only call their methods through superclasses or interfaces, in which case no wrapping/unwrapping is required. When anchored modifiable classes are introduced, however, the control-flow patterns become more complex. Anchored modifiable code is regular application code, and thus can call methods in any other application object. Thus, one anchored modifiable object can well be calling an anchored modifiable object in a different anchored group, which may be remote.

Dynamic wrapping/unwrapping needs to take place in this case. The problem is that an anchored modifiable object has direct references to all its co-anchored objects, but may need to pass those direct references to objects outside the anchored group (either mobile or anchored). For instance, imagine a scenario with co-anchored classes `A` and `B`, and class `C`, packaged in packages `a`, `b`, and `c`, respectively, and anchored on a different site. The original application code may look like the following:

```
package a;
class A {

 b.B b;

 c.C c;
```

```
  void baz () {
    c.foo (b);
    b.B b = c.bar ();
   }
  }

package b;
class B {...}


package c;
class C {
  void foo (b.B b) {...}
  b.B bar () {...}
}
```

If we were to perform a straightforward rewrite of class `A` to refer to `B` directly but to `C` by proxy we would get:

```
package a;
class A {
  b.B b;
  remotecapable.c.C c;
  void baz  () {
   c.foo (b); //incorrectly passing
              //a direct reference to b.B!
   b.B b = c.bar();//incorrectly returning
                   //an indirect ref. to b.B!
  }
}

//proxy for class C
package remotecapable.c;
class C {
  ...
  void foo (remotecapable.b.B b) {...}
  remotecapable.b.B bar () {...}
}
```

As indicated by the comments in the code, this rewrite would result in erroneous bytecodes: direct references are passed to code that expects an indirection and vice versa. A fix could be applied in two places: either at the call site (e.g., the code in class `A` that calls

`c.bar()`) or at the indirection site (i.e., at the proxy `C`, or at some other intermediate object, analogous to the translators we saw in the previous section). The translators of the previous section do the wrapping/unwrapping at the indirection site. Unfortunately this solution is not applicable here. If we were to do the wrapping/unwrapping inside the proxy, the proxy for `C` would look like:

```
// This is imaginary code!
//Irrelevant details (e.g., exception handling) omitted
package remotecapable.c;
class C {
 C__interface ref;
 ...


 // used when caller is outside B's anchored group
 void foo (remotecapable.b.B b) {
  ref.foo ((b.B) Runtime.unwrap(b));
 }
 // used when caller is in B's anchored group
 void foo (b.B b) {
  ref.foo((remotecapable.b.B) Runtime.wrap(b));
 }
 // used when caller is outside B's anchored group
 remotecapable.b.B bar() {
  return ref.bar();
 }
 // used when caller is in B's anchored group
 b.B bar() {
  return ((b.B) Runtime.unwrap(ref.bar());
 }
}
```

Unfortunately, the last two methods differ only in their return type, thus overloading cannot be used to resolve a call to `bar`. This is why a call-site rewrite is required. Since J-Orchestra operates at the bytecode level, this action is not trivial. We need to analyze the bytecode, reconstruct argument types, see if a conversion is necessary, and insert code to

wrap and unwrap objects. The resulting code for our example class `A` is shown below (in source code form, for ease of exposition).

```
package a;
class A {
 b.B b;
 remotecapable.c.C c;


 void baz  () {
  //wrap b in the call to foo
  c.foo ((remotecapable.b.B)Runtime.wrap (b));
  //unwrap b after the call to bar
  b.B b = (b.B) Runtime.unwrap (c.bar());
 }
}
```

A special case of the above problem is self-reference. An object always refers to itself (`this`) directly. If it attempts to pass such references outside its anchored group (or, in the case of a mobile object, to any other object) the reference should be wrapped.

### 4.5.3 Placement Policy Based On Creation Site

The class-based distribution of J-Orchestra is powerful and useful enough for most application scenarios. Using a class as a distribution unit makes assigning classes (or groups of classes) to their destination network sites manageable even for medium to large applications. However, sometimes a distribution policy that is more fine-grained than class-level can become necessary. For example, a meaningful distribution might require placing different instances of the same class on different network sites.

The J-Orchestra creation site placement policy provides an approach that enables such placement. This advanced feature allows the user to distinguish between different instances of the same class, based on the points in the program at which these instances are

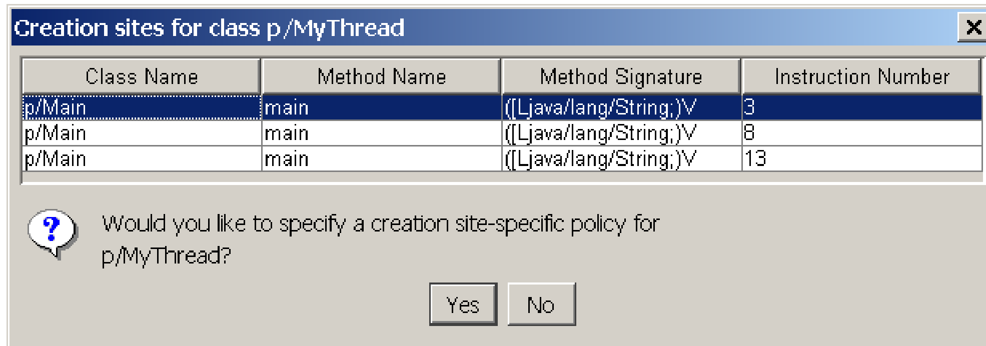


**Figure 4-6:** The results of a query on the creation sites of class p.MyThread.

instantiated. What complicates the implementation of this feature is that because J-Orchestra operates at the bytecode level, source code can not be used to identify such instantiation points.

To demonstrate the creation site placement policy, let us consider the following example. Class `p.MyThread` extends a systems class `java.lang.Thread`, and class `p.Main` has a method `main` that instantiates and calls method `start` on three instances of `p.MyThread` as follows:

```
public static void main (String args[]) {

  p.MyThread thread1 = new p.MyThread (“Thread #1”);
  p.MyThread thread2 = new p.MyThread (“Thread #2”);
  p.MyThread thread3 = new p.MyThread (“Thread #3”);

  thread1.start();
  thread2.start();
  thread3.start();
}
```

Under the standard J-Orchestra partitioning, the classification heuristic would classify class `p.MyThread` as anchored (i.e., it controls threading, a native platform-specific resource), and all its instances would have to be created on the same network site. However, the user can override the classification results by using the creation site specific placement policy.[4] To accomplish that, the user first inquires about the points in the code at which the instances of `p.MyThread` are instantiated. Figure 4-6 shows a GUI dialog box through which the system displays the requested information. As one can see in the dialog box, the system uniquely identifies a creation site by listing its locations, each of which consisting of a class, a method, a method signature, and an instruction number. While the first three parts of a location are self explanatory, the instruction number is simply a heuristic. Because J-Orchestra operates at the bytecode level, the system uses the index of the constructor call instruction in the sequence of the method's bytecodes. After the user chooses a creation site specific policy for a particular class, all creation locations of a class become separate distribution entities that are added to the main tree view of the J-Orchestra. The user can subsequently assign these new distribution entities to separate destination network sites.

### 4.5.4 Object Mobility

One of the ways in which the advanced J-Orchestra user can tune partitioned applications to improve distributed performance is through the use of mobility policies. Object mobility can significantly affect the performance of a distributed application. Mobile objects can exploit application locality and eliminate the need for network communication.

---

4. Such overriding is done at the user's own risk, for thread objects that might be passed to native code will no longer be "anchored" on the same site.

Apart from the creation site placement policy, mobility is the only other mechanism in J-Orchestra that enables per-instance instead of per-class treatment. That is, two objects of the same mobile class can behave entirely differently at run-time based on their uses (i.e., to which methods they are passed as parameters, etc.). Object mobility in J-Orchestra is synchronous: objects move in response to method calls. J-Orchestra supports three object moving scenarios: moving a parameter of a remote method call to the site of the call, moving the return value of a remote method call to the site of the caller, and moving "this" object to the site of the call. In terms of design, our object migration policies are similar to what is commonly found in the mobile objects literature [11][39]. In terms of mechanisms, our implementation bears many similarities to the one in JavaParty [31].

Specifically, J-Orchestra supports mobility through a programming interface and runtime services. Recall that J-Orchestra proxies are generated in source code form. This makes it fairly straightforward to generate additional mobility-specific methods in mobile classes proxies. The user can then use these generated methods as primitives for specifying various mobility scenarios. In addition, each mobile proxy contains a data member of type `MigrationSchema`, which specifies how the object pointed to by the proxy should move. The default value of `MigrationSchema` is `by-reference`, which means that an RMI stub is sent whenever a proxy is passed as a parameter or returned as a result of a remote method call. Mobile proxies enable flexible migration policies by implementing their own serialization. Assigning the value `by-move` to the `MigrationSchema` of a mobile proxy will have the object to which it is pointing move to a remote site. The following generated methods in mobile proxies can be used to specify mobility policies for moving a parameter

of a remote method call to the site of the call and moving the return value of a remote method call to the site of the caller.

```
private MigrationSchema _migrationSchema;
public void setMigrationSchema (MigrationSchema schema)
{...}

public MigrationSchema getMigrationSchema () { ... }


//Overwrite standard serialization behavior
public void writeExternal (ObjectOutput out)
                                   throws IOException {

 Marshaller.marshall(out, this);

}

public void readExternal (ObjectInput in)
           throws IOException, ClassNotFoundException {

 Marshaller.unmarshall(in, this);

}
```

The code below is a (slightly simplified) example of specifying that the parameter `p` of the remote method `foo` should move when the remote method invocation takes place.

```
//proxy method; P is a proxy of a mobile class
public void foo (P p) {
 try {
  //the object pointed by p will move to the site
  //of the method foo, unless p and foo are
  //already collocated.
  p.getMigrationSchema().setByMove();
  //the migration will take place during
  //the serialization of p as part of
  //the invocation of foo.
  _ref.foo (p);

 } catch (RemoteException e) {...}
}
```

The J-Orchestra mobility API contains the following two methods, which can be used to move "this" object (i.e., the one pointed to by the mobile proxy) to and from the site of a remote method invocation.

```
public void moveToRemoteSite (ObjectFactory remoteFac)
{...}

public void moveFromRemoteSite (ObjectFactory remoteFac)
{...}
```

The code below demonstrates how the user can modify the proxy to specify that "this" object should temporarily move over to the local machine to invoke method `bar` locally.

```
//proxy method
public void bar () {
 try {
  ObjectFactory remoteObjectFactory =
             getObjectFactory("SomeSymbolicFactoryName");

  //moves _ref from the remote site, identified by
  //remoteObjectFactory, to the local machine
  moveFromRemoteSite(remoteObjectFactory);
  //execute the call locally
  _ref.bar();
  //moves _ref back to the remote site
  moveToRemoteSite(remoteObjectFactory);

 } catch (RemoteException e) {...}
}
```

One element of the runtime support for mobility in J-Orchestra is the `Marshaller` class, which enables mobility at serialization time. Another important piece of the runtime functionality preserves the "at most one proxy per site" invariant. Because proxies contain unique identifiers, when unserializing a proxy at a remote site, the runtime service checks

whether a proxy with the same unique identifier already exists; if the answer is affirmative, the existing proxy is used instead of instantiating a new one.

An object that is being moved might contain some embedded proxies to other objects, transitively reachable from it. This presents some interesting opportunities for specifying complex mobility scenarios. For example, if object `P` moves, move also objects `Q` and `R`, if they are transitively reachable from it. The existing J-Orchestra infrastructure can be easily extended to support such mobility scenarios, and we would like to pursue this as a possible future work direction.

## 4.6 Dealing with Concurrency and Synchronization

One of the primary design goals of J-Orchestra is to be able to run partitioned programs with standard Java middleware. However, Java middleware mechanisms, such as Java RMI or CORBA implementations, do not support thread coordination over the network: synchronizing on remote objects does not work correctly, and thread identity is not preserved for executions spanning multiple machines. Prior approaches to dealing with the problem suffer from one of two weaknesses: either they require a new middleware mechanism, or they add overhead to the execution to propagate a thread identifier through all method calls. Therefore, these weaknesses leave the existing approaches unable to meet the design goals of J-Orchestra, necessitating a new approach that should work with an unmodified middleware implementation efficiently. We next describe the design, implementation, and evaluation of the J-Orchestra approach to this problem.

### 4.6.1 Overview and Existing Approaches

J-Orchestra enables Java thread synchronization in a distributed setting. This mechanism addresses monitor-style synchronization (mutexes and condition variables), which is well-suited for a distributed threads model. (This is in contrast to low-level Java synchronization, such as volatile variables and atomic operations, which are better suited for symmetric multiprocessor machines.)

This solution is not the first in this design space. Past solutions fall in two different camps. A representative of the first camp is the approach of Haumacher et al. [30], which proposes a replacement of Java RMI that maintains correct multithreaded execution over the network. If employing special-purpose middleware is acceptable, this approach is sufficient. Nevertheless, it would not be suitable for J-Orchestra, which has the ability to use standard middleware as one of its primary design objectives. In general, it is often not desirable to move away from standard middleware, for reasons of portability and ease of deployment. Therefore, the second camp, represented by the work of Weyns, Truyen, and Verbaeten [98], advocates transforming the client application instead of replacing the middleware. Unfortunately, clients (i.e., callers) of a method do not know whether its implementation is local or remote. Thus, to support thread identity over the network, *all* method calls in an application need to be automatically re-written to pass one extra parameter—the thread identifier. This imposes both space and time overhead: extra code is needed to propagate thread identifiers, and adding an extra argument to every call incurs a run-time cost. Weyns, Truyen, and Verbaeten [98] quantify this cost to about 3% of the total execution time of an application. Using more representative macro-benchmarks (from the SPEC JVM suite) we found the cost to be between 5.5 and 12% of the total execution time. A secondary

disadvantage of the approach is that the transformation becomes complex when application functionality can be called by native system code, as in the case of application classes implementing a Java system interface.

J-Orchestra implements a technique that addresses both the problem of portability and the problem of performance. This technique follows the main lines of the approach of Weyns, Truyen, and Verbaeten: it replaces all monitor operations in the bytecode (such as `monitorenter`, `monitorexit`, `Object.wait`) with calls to operations of J-Orchestra distribution-aware synchronization library. Nevertheless, it avoids instrumenting every method call with an extra argument. Instead, it performs a bytecode transformation on the generated RMI stubs. The transformation is general and portable: almost every RPC-style middleware mechanism needs to generate stubs for the remotely invokable methods. By transforming those when needed, it can propagate thread identity information for all remote invocations, without unnecessarily burdening local invocations. This approach also has the advantage of simplicity with respect to native system code. Finally, the J-Orchestra implementation of the approach is fine-tuned, making the total overhead of synchronization be negligible (below 4% overhead even for empty methods and no network cost).

### 4.6.2 Distributed Synchronization Complications

Modern mainstream languages such as Java or C# have built-in support for concurrency. Specifically, Java provides the class `java.lang.Thread` for creating and managing concurrency, monitor methods `Object.wait`, `Object.notify`, and `Object.notifyAll` for managing state dependence, and `synchronized` methods and code blocks for maintaining exclusion among multiple concurrent activities. (An excellent reference for multithreading in Java is Lea's textbook [47].)

Concurrency constructs usually do not interact correctly with middleware implementations, however. In particular, Java RMI does not propagate synchronization operations to remote objects and does not maintain thread identity across different machines.

To see the first problem, consider a Java object `obj` that implements a `Remote` interface `RI` (i.e., a Java interface `RI` that extends `java.rmi.Remote`). Such an object is remotely accessible through the `RI` interface. That is, if a client holds an interface reference `r_ri` that points to `obj`, then the client can call methods on `obj`, even though it is located on a different machine. The implementation of such remote access is the standard RPC middleware technique: the client is really holding an indirect reference to `obj`. Reference `r_ri` points to a local RMI "stub" object on the client machine. The stub serves as an intermediary and is responsible for propagating method calls to the `obj` object. What happens when a monitor operation is called on the remote object, however? There are two distinct cases: Java calls monitor operations (locking and unlocking a mutex) implicitly when a method labeled `synchronized` is invoked and when it returns. This case is handled correctly through RMI, since the stub will propagate the call of a synchronized remote method to the correct site. Nevertheless, all other monitor operations are not handled correctly by RMI. For instance, a `synchronized` block of code in Java corresponds to an explicit mutex lock operation. The mutex can be the one associated with any Java object. Thus, when clients try to explicitly synchronize on a remote object, they end up synchronizing on its stub object instead. This does not allow threads on different machines to synchronize using remote objects: one thread could be blocked or waiting on the real object `obj`, while the other thread may be trying to synchronize on the stub instead of on the `obj` object. Similar problems exist for all other monitor operations. For instance, RMI cannot be used to

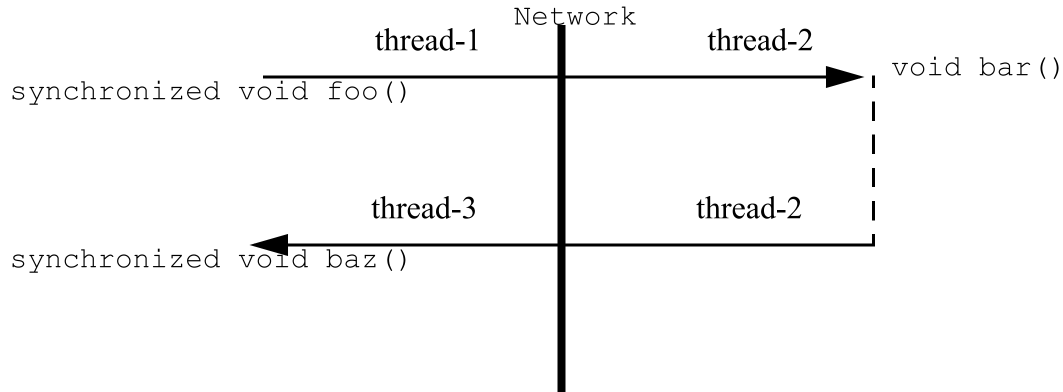


**Figure 4-7:** The zigzag deadlock problem in Java RMI.

propagate monitor operations such as `Object.wait`, `Object.notify`, over the network. The reason is that these operations cannot be indirected: they are declared in class `Object` to be `final`, which means that the methods can not be overridden in subclasses that implement the `Remote` interfaces required by RMI.

The second problem concerns preserving thread identities in remote calls. The Java RMI runtime starts a new thread for each incoming remote call. Thus, a thread performing a remote call has no memory of its identity in the system. Figure 4-7 demonstrates the so-called "zigzag deadlock problem", common in distributed synchronization. Conceptually, methods `foo`, `bar`, and `baz` are all executed in the same thread—but the location of method `bar` happens to be on a remote machine. In actual RMI execution, thread-1 will block until `bar`'s remote invocation completes, and the RMI runtime will start a new thread for the remote invocations of `bar` and `baz`. Nevertheless, when `baz` is called, the monitor associated with thread-1 denies entry to thread-3: the system does not recognize that thread-3 is just handling the control flow of thread-1 after it has gone through a remote machine. If no special care is taken, a deadlock condition occurs.

### 4.6.3 Solution: Distribution-Aware Synchronization

As we saw, any solution for preserving the centralized concurrency and synchronization semantics in a distributed environment must deal with two issues: each remote method call can be executed on a new thread, and standard monitor methods such as `Object.wait`, `Object.notify`, and `synchronized` blocks can become invalid when distribution takes place. Taking these issues into account, we maintain per-site "thread id equivalence classes," which are updated as execution crosses the network boundary; and at the bytecode level, we replace all the standard synchronization constructs with the corresponding method calls to a per-site synchronization library. This synchronization library emulates the behavior of the monitor methods, such as `monitorenter`, `monitorexit`, `Object.wait`, `Object.notify`, and `Object.notifyAll`, by using the thread id equivalence classes. Furthermore, these synchronization library methods, unlike the `final` methods in class `Object` that they replace, get correctly propagated over the network using RMI when necessary so that they execute on the network site of the object associated with the monitor.

In more detail, our approach consists of the following steps:

- Every instance of a monitor operation in the bytecode of the application is replaced, using bytecode rewriting, by a call to our own synchronization library, which emulates the monitor-style synchronization primitives of Java
- Our library operations check whether the target of the monitor operation is a local object or an RMI stub. In the former case, the library calls its local monitor operation. In the latter case, an RMI call to a remote site is used to invoke the appropriate library operation on that site. This solves the problem of propagating monitor operations over the network. We also apply a compile-time optimization to this step: using a simple static

analysis, we determine whether the target of the monitor operation is an object that is known statically to be on the current site. This is the case for monitor operations on the `this` reference, as well as other objects of anchored types that J-Orchestra guarantees will be on the same site throughout the execution. If we know statically that the object is local, we avoid the runtime test and instead call a local synchronization operation.

- Every remote RMI call, whether on a synchronized method or not, is extended to include an extra parameter. The instrumentation of remote calls is done by bytecode transformation of the RMI stub classes. The extra parameter holds the thread equivalence class for the current calling thread. Our library operations emulate the Java synchronization primitives but do not use the current, machine-specific thread id to identify a thread. Instead, a mapping is kept between threads and their equivalence classes and two threads are considered the same if they map to the same equivalence class. Since an equivalence class can be represented by any of its members, our current representation of equivalence classes is compact: we keep a combination of the first thread id to join the equivalence class and an id for the machine where this thread runs. This approach solves the problem of maintaining thread identity over the network.

We illustrate the above steps with examples that show how they solve each of the two problems identified earlier. We first examine the problem of propagating monitor operations over the network. Consider a method as follows:

```
//original code
void foo (Object some_remote_object) {

 this.wait();
 ...
 some_remote_object.notify();
 ...


}
```

At the bytecode level, method `foo` will have a body that looks like:

```
//bytecode
aload_0
invokevirtual     java.lang.Object.wait
...
aload_1
invokevirtual     java.lang.Object.notify
...
```

Our rewrite will statically detect that the first monitor operation (`wait`) is local, as it is called on the current object itself (`this`). The second monitor operation, however, is (potentially) remote and needs to be redirected to its target machine using an RMI call. The result is shown below:

```
//rewritten bytecode
aload_0
//dispatched locally
invokestatic jorchestra.runtime.distthreads.wait_
...
aload_1
//get thread equivalence info from runtime
invokestatic
jorchestra.runtime.ThreadInfo.getThreadEqClass
//dispatched through RMI;
//all remote interfaces extend DistSyncSupporter
invokeinterface jorchestra.lang.DistSynchSupporter.notify_
...
```

(The last instruction is an interface call, which implies that each remote object needs to support monitor methods, such as `notify_`. This may seem to result in code bloat at first, but our transformation adds these methods to the topmost class of each inheritance hierarchy in an application, thus minimizing the space overhead.)

Let's now consider the second problem: maintaining thread identity over the network. Figure 4-8 demonstrates how using the thread id equivalence classes can solve the "zigzag deadlock problem" presented above. These thread id equivalence classes enable

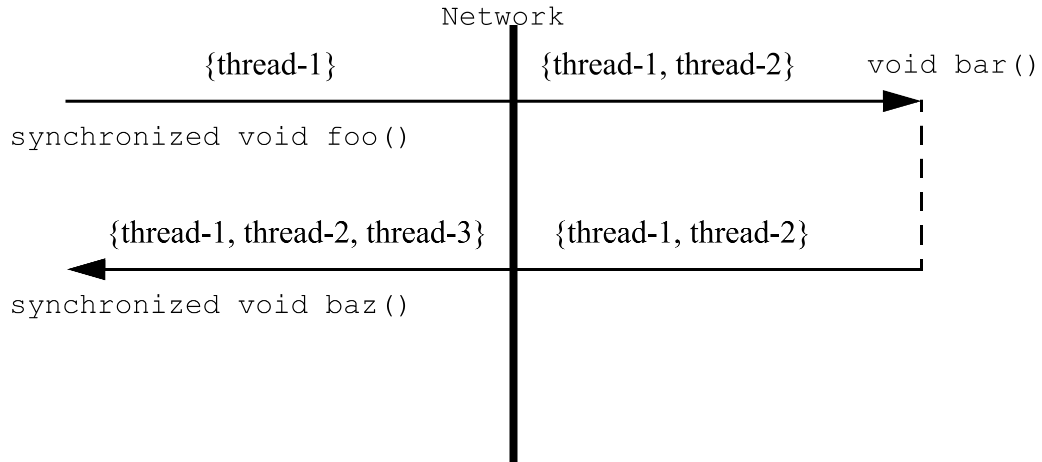


**Figure 4-8:** Using thread id equivalence classes to solve the "zigzag deadlock problem" in Java RMI.

our custom monitor operations to treat all threads within the same equivalence class as the same thread. (We illustrate the equivalence class by listing all its members in the figure, but, as mentioned earlier, in the actual implementation only a single token that identifies the equivalence class is passed across the network.) More specifically, our synchronization library is currently implemented using regular Java mutexes and condition variables. For instance, the following code segment (slightly simplified) shows how the library emulates the behavior of the bytecode instruction `monitorenter`. (For readers familiar with monitor-style concurrent programming, our implementation should look straightforward.) The functionality is split into two methods: the static method `monitorenter` finds or creates the corresponding `Monitor` object associated with a given object: our library keeps its own mapping between objects and their monitors. The member method `enter` of class `Monitor` causes threads that are not in the equivalence class of the holder thread to wait until the monitor is unlocked.

```
public static void monitorenter (Object o) {
 Monitor this_m = null;
 synchronized (Monitor.class) {
  this_m = (Monitor)_objectToMonitor.get(o);
  if (this_m == null) {
   this_m = new Monitor();
   _objectToMonitor.put(o, this_m);
  }
 } //synchronized
 this_m.enter();
}


private synchronized void enter () {
 while (_timesLocked != 0 &&
        curThreadEqClass != _holderThreadId)
  try { wait(); } catch(InterruptedException e) {...}

  if (_timesLocked == 0) {
  _holderThreadId = getThreadID();
 }
 _timesLocked++;
}
```

The complexity of maintaining thread equivalence classes determines the overall efficiency of the solution. The key to efficiency is to update the thread equivalence classes only when necessary—that is, when the execution of a program crosses the network boundary. Adding the logic for updating equivalence classes at the beginning of every remote method is not the appropriate solution: in many instances, remote methods can be invoked locally within the same JVM. In these cases, adding any additional code for maintaining equivalence classes to the remote methods themselves would be unnecessary and detrimental to performance. In contrast, our solution is based on the following observation: the program execution will cross the network boundary only after it enters a method in an RMI stub. Thus, RMI stubs are the best location for updating the thread id equivalence classes on the client site of a remote call.

Adding custom logic to RMI stubs can be done by modifying the RMI compiler, but this would negate our goal of portability. Therefore, we use bytecode engineering on standard RMI stubs to retrofit their bytecode so that they include the logic for updating the thread id equivalence classes. This is done completely transparently relative to the RMI runtime by adding special delegate methods that look like regular remote methods, as shown in the following code example. To ensure maximum efficiency, we pack the thread equivalence class representation into a long integer, in which the less significant and the most significant 4 bytes store the first thread id to join the equivalence class and the machine where this thread runs, respectively. This compact representation significantly reduces the overhead imposed on the remote method calls, as we demonstrate later on. Although all the changes are applied to the bytecode directly, we use source code for ease of exposition.

```
//Original RMI stub: two remote methods foo and bar
class A_Stub ... {
 ...
 public void foo (int i) throws RemoteException {...}
 public int bar () throws RemoteException {...}
}

//Retrofitted RMI stub
class A_Stub ... {
 ...
 public void foo (int i) throws RemoteException {
   foo__tec (Runtime.getThreadEqClass(), i);
 }

 public void foo__tec (long tec, int i)
                                  throws RemoteException
 {...}

 public int bar () throws RemoteException {
   return bar__tec (Runtime.getThreadEqClass());
 }
```

```
  public int bar__tec (long tec) throws RemoteException
  {...}
}
```

Remote classes on the callee site provide symmetrical delegate methods that update the thread id equivalence classes information according to the received `long` parameter, prior to calling the actual methods. Therefore, having two different versions for each remote method (with the delegate method calling the actual one) makes the change transparent to the rest of the application: neither the caller of a remote method nor its implementor need to be aware of the extra parameter. Remote methods can still be invoked directly (i.e., not through RMI but from code on the same network site) and in this case they do not incur any overhead associated with maintaining the thread equivalence information.

### 4.6.4 Benefits of the Approach

The two main existing approaches to the problem of maintaining the centralized Java concurrency and synchronization semantics in a distributed environment have involved either using custom middleware [30] or making universal changes to the distributed program [98]. We argue next that our technique is a good fit for J-Orchestra, being more portable than using custom middleware and more efficient than a universal rewrite of the distributed program. Finally, we quantify the overhead of our approach and show that our implementation is indeed very efficient.

#### 4.6.4.1 Portability

In our context, a solution for preserving the centralized concurrency and synchronization semantics in a distributed environment is only as useful as it is portable. A solution is portable if it applies to different versions of the same middleware (e.g., past and future

versions of Java RMI) and to different middleware mechanisms such as CORBA and .NET Remoting. Our approach is both simple and portable to other middleware mechanisms, because it is completely orthogonal to other middleware functionality: We rely on bytecode engineering, which allows transformations without source code access, and on adding a small set of runtime classes to each network node of a distributed application. The key to our transformation is the existence of a client stub that redirects local calls to a remote site. Using client stubs is an almost universal technique in modern middleware mechanisms. Even in the case when these stubs are generated dynamically, our technique is applicable, as long as it is employed at class load time.

For example, our bytecode instrumentation can operate on CORBA stubs as well as it does on RMI ones. Our stub transformations simply consist of adding delegate methods (one for each client-accessible remote method) that take an extra thread equivalence parameter. Thus, no matter how complex the logic of the stub methods is, we would apply to them the same simple set of transformations.

Some middleware mechanisms such as the first version of Java RMI also use server-side stubs (a.k.a. *skeletons*) that dispatch the actual methods. Instead of presenting complications, skeletons would even make our approach easier. The skeleton methods are perfect for performing our server-side transformations, as we can take advantage of the fact that the program execution has certainly crossed the network boundary if it entered a method in a skeleton. Furthermore, having skeletons to operate on would eliminate the need to change the bytecodes of the remote classes. Finally, the same argument of the simplicity of our stub transformations being independent of the complexity of the stub code itself equally applies to the skeleton transformations.

In a sense, our approach can be seen as adding an orthogonal piece of functionality (concurrency control) to existing distribution middleware. In this sense, one can argue that the technique has an aspect-oriented flavor.

#### 4.6.4.2 The Cost of Universal Extra Arguments

Our approach eliminates both the runtime and the complexity overheads of the closest past techniques in the literature. Weyns, Truyen, and Verbaeten [98][99] have advocated the use of a bytecode transformation approach to correctly maintain thread identity over the network. Their technique is occasionally criticized as "incur[ring] great runtime overhead" [30]. The reason is that, since clients do not know whether a method they call is local or remote, every method in the application is extended with an extra argument—the current thread id—that it needs to propagate to its callees. Weyns et al. argue that the overhead is acceptable and present limited measurements where the overhead of maintaining distributed thread identity is around 3% of the total execution time. Below we present more representative measurements that put this cost at between 5.5 and 12%. A second cost that has not been evaluated, however, is that of complexity: adding an extra parameter to all method calls is hard when some clients cannot be modified because, e.g., they are in native code form or access the method through reflection. In these cases a correct application of the Weyns et al. transformation would incur a lot of complexity. This complexity is eliminated with our approach.

It is clear that some run-time overhead will be incurred if an extra argument is added and propagated to every method in an application. To see the range of overhead, we wrote a simple micro-benchmark, in which each method call performs one integer arithmetic operation, two comparisons and two (recursive) calls. Then we measured the overhead of

adding one extra parameter to each method call. Figure 4-1 shows the results of this benchmark. For methods with 1-5 integer arguments we measure their execution time with one extra reference argument propagated in all calls. As seen, the overhead varies unpredictably but ranges from 5.9 to 12.7%.

**Table 4-1.** Micro-benchmark: overhead of method calls with one extra argument.

| #params | 1 (base) | 1+1 | 2+1 | 3+1 | 4+1 | 5+1 |
|---|---|---|---|---|---|---|
| Execution time (sec) for 10^8 calls | 1.945 | 2.059 | 2.238 | 2.523 | 2.691 | 2.916 |
| Slowdown relative to previous | - | 5.9% | 8.7% | 12.7% | 6.7% | 8.4% |

Nevertheless, it is hard to get a representative view of this overhead from micro-benchmarks, especially when running under a just-in-time compilation model. Therefore, we concentrated on measuring the cost on realistic applications. As our macro-benchmarks, we used applications from the SPEC JVM benchmark suite. Of course, some of the applications we measured may not be multithreaded, but their method calling patterns should be representative of multithreaded applications, as well.

We used bytecode instrumentation to add an extra reference argument to all methods and measured the overhead of passing this extra parameter. In the process of instrumenting realistic applications, we discovered the complexity problems outlined earlier. The task of adding an extra parameter is only possible when all clients can be modified by the transformation. Nevertheless, all realistic Java applications present examples where clients will not be modifiable. An application class can be implementing a system interface, making native Java system code a potential client of the class's methods. For instance, using class frameworks, such as AWT, Swing, or Applets, entails extending the classes provided by such frameworks and overriding some methods with the goal of customizing the application's

behavior. Consider, for example, a system interface `java.awt.TextListener`, which has a single method `void textValueChanged (TextEvent e)`. A non-abstract application class extending this interface has to provide an implementation of this method. It is impossible to add an extra parameter to the method `textValueChanged` since it would prevent the class from being used with AWT. Similarly a Java applet overrides methods `init`, `start`, and `stop` that are called by Web browsers hosting the applet. Adding an extra argument to these methods in an applet would invalidate it. These issues can be addressed by careful analysis of the application and potentially maintaining two interfaces (one original, one extended with an extra parameter). Nevertheless, this would result in code bloat, which could further hinder performance.

Since we were only interested in quantifying the potential overhead of adding and maintaining an extra method parameter, we sidestepped the complexity problems by avoiding the extra parameter for methods that could be potentially called by native code clients. Instead of changing the signatures of such methods so that they would take an extra parameter, we created the extra argument as a local variable that was passed to all the callees of the method. The local variable is never initialized to a useful value, so no artificial overhead is added by this approach. This means that our measurements are slightly conservative: we do not really measure the cost of correctly maintaining an extra thread identity argument but instead conservatively estimate the cost of passing one extra reference parameter around. Maintaining the correct value of this reference parameter, however, may require some extra code or interface duplication, which may make performance slightly worse than what we measured.

Another complication concerns the use of Java reflection for invoking methods, which makes adding an extra argument to such methods impossible. In fact, we could not correctly instrument all the applications in the SPEC JVM suite, exactly because some of them use reflection heavily and we would need to modify such uses by hand.

The results of our measurements appear in Table 4-2. The table shows total execution time for four benchmarks (compress—a compression utility, javac—the Java compiler, mtrt—a multithreaded ray-tracer, and jess—an expert system) in both the original and instrumented versions, as well as the slowdown expressed as the percentage of the differences between the two versions, ranging between 5.5 and 12%. The measurements were on a 600MHz Pentium III laptop, running JDK 1.4.

**Table 4-2.** Macro-benchmarks: cost of a universal extra argument.

| Benchmark | compress | javac | mtrt | jess |
|---|---|---|---|---|
| Original version (sec) | 22.403 | 19.74 | 6.82 | 8.55 |
| Instrumented version (sec) | 23.644 | 21.18 | 7.49 | 9.58 |
| Slowdown | 5.54% | 7.31% | 9.85% | 12.05% |

The best way to interpret these results is as the overhead of pure computation (without communication) that these programs would incur under the Weyns et al. technique if they were to be partitioned with J-Orchestra so that their parts would run correctly on distinct machines. We see, for instance, that running jess over a network would incur an overhead of 12% in extra computation, just to ensure the correctness of the execution under multiple threads. Our approach eliminates this overhead completely: overhead is only incurred when actual communication over distinct address spaces takes place. As we show next, this overhead is minuscule relative to total execution time, even for an infinitely fast network and no computation performed by remote methods.

#### 4.6.4.3 Maintaining Thread Equivalence Classes Is Cheap

Maintaining thread equivalence classes, which consists of obtaining, propagating, and updating them, constitutes the overhead of our approach. In other words, to maintain the thread equivalence classes correctly, each retrofitted remote method invocation includes one extra local method call on the client side to obtain the current class, an extra argument to propagate it over the network, and another local method call on the server side to update it. The two extra local calls, which obtain and update thread equivalence classes, incur virtually no overhead, having a hash table lookup as their most expensive operation and causing no network communication. Thus, the cost of propagating the thread equivalence class as an extra argument in each remote method call constitutes the bulk of our overhead.

In order to minimize this overhead, we experimented with different thread equivalence classes' representations. We performed preliminary experiments which showed that the representation does matter: the cost of passing an extra reference argument (any subclass of `java.lang.Object` in Java) over RMI can be high, resulting in as much as 50% slowdown in the worst case. This happens because RMI accomplishes the marshalling/unmarshalling of reference parameters via Java serialization, which involves dynamic memory allocation and the use of reflection. Such measurements led us to implement the packed representation of thread equivalence class information into a long integer, as described earlier. A `long` is a primitive type in Java, hence the additional cost of passing one over the network became negligible.

To quantify the overall worst-case overhead of our approach, we ran several microbenchmarks, measuring total execution time taken by invoking empty remote meth-

ods with zero, one `java.lang.String`, and two `java.lang.String` parameters. Each remote method call was performed 10^6 times. The base line shows the numbers for regular uninstrumented RMI calls. To measure the pure overhead of our approach, we used an unrealistic setting of collocating the client and the server on the same machine, thus eliminating all the costs of network communication. The measurements were on a 2386MHz Pentium IV, running JDK 1.4. The results of our measurements appear in Table 4-3.

**Table 4-3.** Overhead of Maintaining Thread Equivalence Classes

| No. of Params | Base Line (ms) | Maintaining Thread Equivalence Classes (ms) | Overhead (%) |
|---|---|---|---|
| 0 | 145,328 | 150,937 | 3.86% |
| 1 | 164,141 | 166,219 | 1.27% |
| 2 | 167,984 | 168,844 | 0.51% |

Since the remote methods in this benchmark did not perform any operations, the numbers show the time spent exclusively on invoking the methods. While the overhead is approaching 4% for the remote method without any parameters, it diminishes gradually to half a percent for the method taking two parameters. Of course, our settings for this benchmark are strictly worst-case—had the client and the server been separated by a network or had the remote methods performed any operations, the overhead would strictly decrease.

### 4.6.5 Discussion

As we mentioned briefly earlier, the J-Orchestra distributed synchronization approach only supports monitor-style concurrency control. This is a standard application-level concurrency control facility in Java, but it is not the only one and the language has currently evolved to better support other models. For example, high-performance applications may use `volatile` variables instead of explicit locking. In fact, use of non-monitor-

style synchronization in Java will probably become more popular in the future. The JSR-166 specification has standardized many concurrent data structures and atomic operations in Java 5. Although our technique does not support all the tools for managing concurrency in the Java language, this is not so much a shortcoming as it is a reasonable design choice. Low-level concurrency mechanisms (volatile variables and their derivatives) are useful for synchronization in a single memory space. Their purpose is to achieve optimized performance for symmetric multiprocessor machines. In contrast, our approach deals with correct synchronization over middleware—i.e., it explicitly addresses distributed memory, resulting from partitioning. Programs partitioned with J-Orchestra are likely to be deployed on a cluster or even a more loosely coupled network of machines. In this setting, monitor-style synchronization makes perfect sense.

On the other hand, in the future we can use the lower-level Java concurrency control mechanisms to optimize the J-Orchestra runtime synchronization library for emulating Java monitors. As we saw in Section 4.6.3, our current library is itself implemented using monitor-style programming (`synchronized` blocks, `Object.wait`, etc.). With the use of optimized low-level implementation techniques, we can gain in efficiency. We believe it is unlikely, however, that such a low-level optimization in our library primitives will make a difference for most client applications of our distributed synchronization approach.

Finally, we should mention that our current implementation does not handle all the nuances of Java monitor-style synchronization, but the issue is one of straightforward engineering. Notably, we do not currently propagate `Thread.interrupt` calls to all the nodes that might have threads blocked in an invocation of the `wait` method. Even though it is unlikely that the programs amenable to automatic partitioning would ever use the `inter-`

`rupt` functionality, our design can easily support it. We can replace all the calls to `Thread.interrupt` with calls to our synchronization library, which will obtain the equivalence class of the interrupted thread and then broadcast it to all the nodes of the application. The node (there can be only one) that has a thread in the equivalence class executing the `wait` operation of our library will then stop waiting and the operation will throw the `InterruptedException`.

To summarize, the J-Orchestra technique for correct monitor-style synchronization of distributed programs in Java addresses the lack of coordination between Java concurrency mechanisms and Java middleware. This technique comprehensively solves the problem and combines the best features of past approaches by enabling distributed synchronization that is both portable and efficient.

## 4.7 Appletizing: Partitioning for Specialized Domains

Some domains present interesting opportunities for specializing J-Orchestra partitioning. One such domain is a client-server environment in which the client runs as a Java applet that communicates with the server through RMI. We call this specialization *appletizing*, and its primary purpose is adapting legacy Java code for distributed execution. In our context, the term ‘legacy’ refers to all centralized Java applications, written without distribution in mind, that as part of their evolution need to move parts of their execution functionality to a remote machine. The amount of such legacy code in Java is by no means insignificant with the Java technology being a decade old and four million Java developers worldwide [13].

A large part of what makes Java a language that "allows application developers to write a program once and then be able to run it everywhere on the Internet" [25] are standard distribution technologies over the web. Such Java technologies as applets and servlets have two major advantages: they require little to no explicit distributed programming and they are easily deployable over standard web browsers. Nevertheless, these technologies are inflexible. In the case of applets, a web browser first transfers an applet's code from the server site to the user system and then executes it safely on its JVM, usually in order to draw graphics on the client's screen. In the symmetric case of servlets, code executes entirely on the server, usually in order to access a shared resource such as a database, transmitting only simple inputs and outputs over the network. Therefore, these standard technologies offer a hard-coded answer to the important question of how the distribution should take place, and it is the same for each applet and servlet. Besides these two extremes, one can imagine many other solutions that are customizable for individual programs. A hybrid of the two approaches promises significant flexibility benefits: the programmer can leverage both the resources of the client machine (e.g., graphics, sound, mouse input) and the resources of a server (e.g., shared database, file system, computing power). At the same time, the application will be both safe and efficient: one can benefit from the security guarantees provided by Java applets, while communicating only a small amount of data between the applet and a remote server.

The challenge is to get an approach that runs code both on clients and on a server while avoiding explicit distributed systems development, just like applet and servlet technologies do. Appletizing implements such an approach by semi-automatically transforming a centralized, monolithic Java GUI application into a client-server application, in which the

client runs as a Java applet that communicates with the server through Java RMI. Appletizing builds upon and is a specialization of automatic partitioning with a predefined deployment environment for the resulting client-server applications. Similarly to regular partitioning, appletizing requires no explicit programming or modification to the JVM or its standard runtime classes.

At the same time, the specialized domain makes appletizing more automatic, which required adding several new features to J-Orchestra such as a new static analysis heuristics that automatically assigns classes to the client and the server sites, a profiling heuristic implementation, special bytecode rewrites that adapt certain operations for execution within an applet, and runtime support for the applet/server coordination.

Overall, appletizing offers a unique combination of the following benefits:

- Programming advantages. This includes no-coding distribution and flexibility in writing applications that use complex graphical interfaces and remote resources.
- User deployment advantages. With the client part running as a regular Java applet rather than as a stand-alone distributed application, our approach is accessible to the user via any Java-enabled browser.
- Performance advantages. Appletizing minimizes network traffic through profiling-based object placement and object mobility. This results in transferring less data than when using such remote control technologies as X-Windows.

The advantage of automatic partitioning is that it can put the code near the resource that it controls. In the case of appletizing, partitioning makes it possible to draw graphics locally on the client machine from less data than it takes to transfer the entire graphical representation over the network, while collocating the server resources with the code that controls them. As a special kind of partitioning, appletizing not only offers the same benefits

but also provides a higher degree of automation by enhancing the capacities of several J-Orchestra mechanisms. Next, we describe the specifics of appletizing by detailing the functionality added to static analysis, profiling, bytecode rewriting, and runtime services.

### 4.7.1 Static Analysis for Appletizing

Consider an arbitrary centralized Java AWT/Swing application that we want to transform into a client-server application through appletizing. First, we classify the application's code (both application classes and the referenced JRE system classes) into four distinct groups, as Figure 4-9 demonstrates schematically.

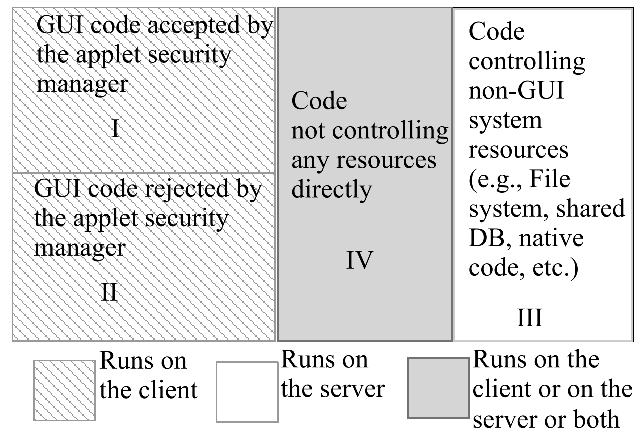


**Figure 4-9:** The appletizing perspective code view of a centralized Java GUI application.

Group I contains the GUI classes that can safely execute within an applet. Group II contains the GUI classes whose code include instructions that the applet security manager prevents from executing within an applet. For example, an applet cannot perform disk I/O. Group III contains the classes that must execute on the server. The classes in this group control various non-GUI system resources that applets are not allowed to access, such as file I/O operations, shared resources (e.g., a database), and native (JNI) code. Group IV contains

the classes that do not control any system resources directly and as such can be placed on either the client or the server sites, purely for performance reasons. Moreover, objects of classes in this group do not have to remain on the same site during the execution of the program: they can migrate on demand, or according to an application-specific pattern. We implemented the analysis of classes for appletizing on top of the standard J-Orchestra type-based "classification" heuristic (Section 4.4) that groups classes whose instances can be accessed by the same native code.

### 4.7.2 Profiling for Appletizing

Having completed the aforementioned classification heuristics, J-Orchestra assigns the classes in groups I, II, and III to the client, client, and server sites, respectively. The classification does not offer any help in assigning the classes in group IV, so the user has to do this manually before the rewriting for appletizing can commence. Deciding on the location of a class just by looking at its name can be a prohibitively difficult task, particularly if no source code is available and the user has only a black-box view of the application. To help the user in determining a good placement, J-Orchestra offers an off-line profiler that reports data exchange statistics among different entities (i.e., anchored groups and mobile classes). Integrated with the profiler is a clustering heuristic that, given some initial locations and the profiling results, determines a good placement for all classes. The heuristic is strictly advisory—the user can override it at will. Our heuristic implements a greedy strategy: start with the given initial placement of a few entities and compute the affinity of each unassigned entity to each of the locations. (Affinity to a location is the amount of data exchanged between the entity and all the entities already assigned to the location combined.) Pick the overall maximum of such affinity, assign the entity that has it to the corresponding location

and repeat until all entities are assigned. In principle, appletizing offers more opportunities than general application partitioning for automation in clustering: optimal clustering for a client-server partitioning can be done in polynomial time, while for 3 or more partitions the problem is NP-hard [24]. In practice we have not yet had the need to replace our heuristic for better placement.

In terms of implementation, the J-Orchestra profiler has evolved through several incarnations. The first profiler worked by instrumenting the Java VM through the JVMPI and JVMDI (Java Virtual Machine Profiling/Debugging Interface) binary interfaces. We found the overheads of this approach to be very high, even for recent VMs that enable compiled execution under debug mode. The reason is the "impedance mismatch" between the profiling code (which is written in C++ and compiled into a dynamic library that instruments the VM) and the Java object layout. Either the C++ code needs to use JNI to access object fields, or the C++ code needs to call a Java library that will use reflection to access fields. We have found both approaches to be much slower (15x) than using bytecode engineering to inject our own profiling code in the application. The profiler rewrite is isomorphic to the J-Orchestra rewrite, except that no distribution is supported—proxies keep track of the amount of data passed instead.

An important issue with profiling concerns the use of off-line vs. on-line profiling. Several systems with goals similar to ours (e.g., Coign [33] and AIDE [56]) use on-line profiling in order to dynamically discover properties of the application and possibly alter partitioning decisions on-the-fly. So far, we have not explored an on-line approach in J-Orchestra, because of its overheads for regular application execution. Since J-Orchestra has no control over the JVM, these overheads can be expected to be higher than in other sys-

tems that explicitly control the runtime environment. Without low-level control, it is hard to keep such overhead to a minimum. Sampling techniques can alleviate the overhead (at the expense of some accuracy) but not eliminate it: some sampling logic has to be executed in each method call, for instance. We hope to explore the on-line profiling direction in the future.

### 4.7.3 Rewriting Bytecode for Appletizing

Once all the classes are assigned to their destination sites, J-Orchestra rewrites the application for appletizing, which augments the regular J-Orchestra rewrite with an additional step that modifies unsafe instructions in GUI classes for executing within an applet. The applet security manager imposes many restrictions on what resources applets can access, and many of these restrictions affect GUI code. J-Orchestra inspects the bytecode of an application for problematic operations and "sanitizes" them for safe execution within an applet. Depending on the nature of an unsafe operation, J-Orchestra uses two different replacement approaches. The first approach replaces an unsafe operation with a safe, semantically similar (if not identical) version of it. For example, an unsafe operation that reads a graphical image from disk gets rewritten with a safe operation that reads the same image from the applet's jar file. The following code fragment demonstrates the rewrite. (We use source code only for ease of exposition—all modifications take place at the bytecode level):

```
//Creates an ImageIcon from the specified file
//will cause a security exception
//when a file on disk is accessed
javax.swing.ImageIcon icon =
           new javax.swing.ImageIcon ("SomeIconFile.gif");
```

```
//Sanitize by replacing with the following safe code
javax.swing.ImageIcon icon =
    new jorchestra.runtime.ImageIcon ("SomeIconFile.gif");

//The implementation of jorchestra.runtime.ImageIcon
//constructor
//(part of J-Orchestra runtime functionality)
public ImageIcon (String fileName) {
 //obtain and pass a URL to the super constructor
 //of javax.swing.ImageIcon
 super (getURL (fileName)); //will safely read the image
                            //from the applets's jar file
}
```

The second approach, replaces an unsafe operation with a semantically different operation. For example, since applets are not allowed to call `System.exit`, this method call gets replaced with a call to the J-Orchestra runtime service that informs the user that they can exit the applet by directing the web browser to another page. Sometimes, replacing an unsafe instruction requires a creative solution. For example, the applet security manager prevents the `setDefaultCloseOperation` method in class `javax.swing.JFrame` from accepting the value `EXIT_ON_CLOSE`. Since we cannot change the code inside `JFrame`, which is a system class, we modify the caller bytecode to pop the potentially unsafe parameter value off the stack and push the safe value `DO_NOTHING_ON_CLOSE` before calling `setDefaultCloseOperation`. The following code snippet demonstrates the specifics of this bytecode replacement.

```
//the following two instructions are inserted
//before every invocation
//of setDefaultCloseOperation method of javax.swing.JFrame

pop //pop whatever value on top of the stack
push 0 //param 0 means DO_NOTHING_ON_CLOSE
invokevirtual javax.swing.JFrame.setDefaultCloseOperation
```

```
//If there are any jumps whose target is
//the invokevirtual instruction,
//they are redirected to the pop instruction instead.
```

Once unsafe instructions in GUI classes have been replaced, J-Orchestra proceeds with its standard rewrite that ends up packaging all the rewritten classes in client and server jar files ready for deployment.

The GUI-intensive nature of appletizing also allows us to perform special-purpose code transformations to optimize remote communication. For instance, knowing the design principles of the Swing/AWT libraries allows us to pass Swing event objects using by-copy semantics. This is done by making event objects implement `java.io.Serializable` and adding a default no arguments constructor if it is not already present. Passing event objects by-copy is typically safe because event listener code commonly uses event objects as read-only objects, since the programming model makes it very difficult to determine in what order event listeners receive events.

Currently the rewrite does not fully maintain the Swing design invariant of having all event-dispatching and painting code execute in a single event-dispatching thread. This can make a graphical application execute incorrectly when partitioned for distributed execution. The problem is that splitting a single-threaded application into a client and server parts creates implicit multithreading. Thus, the server could call client Swing code remotely through RMI on a thread different from the event-dispatching one. Figure 4-10 demonstrates pictorially how this situation could occur. This is a so-called zig-zag calling pattern, in which a GUI calls `someServerMethod` on the server. Then `someServerMethod`, in response, calls back `someGUIOp` method on the client, which is invoked on a new thread, different from the one designated for event dispatching. As we have seen in

Applet site (runs GUI) Network Server Site

*Event-Disp-Thread* *Server-RMI-Thread* someServerMethod(){

someServerMethod();

*Client-RMI-Thread* *Server-RMI-Thread* someGUIOp();

void someGUIOp()

}

**Figure 4-10:** Violating the Swing threading design invariant: `someGUIOp` method is invoked on a thread different than *Event-Disp-Thread,* if no special care is taken.

Section 4.6, this happens because Java RMI does not preserve thread identity for executions spanning multiple machines. The J-Orchestra approach to maintaining concurrency and synchronization over RMI simply treats different threads (e.g., *Event-Disp-Thread*, *Server-RMI-Thread*, and *Client-RMI-Thread* in this case) as belonging to the same equivalence class and as such cannot ensure that all GUI code is executed in a single designated event-dispatching thread. Therefore, appletizing must enable special-purpose handling of Swing code.

A fully general solution would ensure that the event-dispatching thread never blocks while making a remote call. This can be accomplished by spawning a new thread for each remote call made in the event-dispatching thread. Then the event-dispatching thread would just wait for two kinds of events: either for the spawned thread to complete or for newly-arrived GUI events (e.g., screen updates) that need to take place. If the server makes a remote call to a GUI method, the method will just add a new GUI event to a queue and will notify the event-dispatching thread to execute it, while the server is blocked—just as in the

local execution case. We plan to implement this faithful emulation of the local execution semantics in the future. Currently, however, we only offer a simpler, approximate solution that handles a special case of the problem. We also report to the user all instances of GUI methods potentially called by the server part of the application, since we do not transparently guarantee correct execution in all cases.

Specifically, our current solution works only for GUI methods that return `void` and do not change the state of the application in any way other than by producing graphical output. In most cases, when the backend calls the front end GUI, it does so through the so called callback methods that just perform some drawing actions and do not return any values. The current implementation uses the existing Swing facility (`SwingUtilities.invokeLater` method) to enable any thread to request that the event-dispatching thread runs certain code. The following is the code in the translator of some GUI class for method `someGUIOp` from Figure 4-7, executing at the client site:

```
//make all parameters final, to be able to pass them
//to the anonymous inner class
public void someGUIOp (final int param) throws
RemoteException {

 SwingUtilities.invokeLater(new Runnable () {
   public void run () {
     _localObj.someGUIOp(param);
   }
 });
}
```

In other words, whenever a `void`-returning method performs operations using the Swing library (perhaps transitively, through other method calls), we make sure that remote calls of the method result in its delayed execution inside the event dispatching thread. The actual GUI code will be executed only when the remote call returns. We have found this

incomplete solution sufficient for successfully appletizing the applications described in our case studies in Section 5.7.1.

Note that both the current partial solution and a future full emulation of local Swing semantics fit well in our appletizing techniques. Recall the structure of the J-Orchestra indirection approach: classes that are co-anchored on the same site, such as the applet's GUI classes, end up calling each other directly. The server classes, on the other hand, can call client classes (and vice versa) only through a proxy/translator chain. Thus, all events that we need to trap (namely, remote calls inside the event-dispatching thread and remote invocations of a GUI method) are handled through a translator—hence, only the code inside the generated translator classes needs to change. This is simple, as it requires no modification of the existing binary code of the application, and imposes no overhead on the local execution of methods.

### 4.7.4 Runtime Support for Appletizing

Appletizing works with standard Java-enabled browsers that download the applet code from a remote server. To simplify deployment, the downloaded code is packaged into two separate jar files, one containing the application classes that run on the client and the other J-Orchestra runtime classes. In other words, the client of an appletized application does not need to have pre-installed any J-Orchestra runtime classes, as a Java-enabled browser downloads them along with the applet classes. Once the download completes, the J-Orchestra runtime client establishes an RMI connection with the server and then invokes the main method of the application through reflection. The name of the application class that contains the main method along with the URL of the server's RMI service are supplied as applet parameters in an automatically generated HTML file. Figure 4-11 shows such an

```
<html>
 <head><title>Jarminator run as a J-Orchestra Applet</title>
 </head>
 <body>
 <APPLET WIDTH=1 HEIGHT=1
   CODE="jorchestra/runtime/applet/Applet.class"
   ARCHIVE="jarminator.jar, jorchestra.jar" >
   <PARAM NAME="CLASSNAME"
      VALUE="remotecapable.net.weird173.jarminator.Jarminator">
  <PARAM NAME="CLIENT_NODE_NAME" VALUE="jarminator_client">
  <PARAM NAME="SERVER_NODE_NAME" VALUE="jarminator_server">
 </APPLET>
 </body>
</html>
```

**Figure 4-11:** An automatically generated HTML file for deploying the appletized Jarminator application.

HTML file for one of our case studies, which is discussed in-detail in Section 5.7.1. This arrangement allows hosting multiple J-Orchestra applets on the same server that can share the same set of runtime classes. In addition, multiple clients can simultaneously run the same applet, but they will also spawn distinct server components. Our approach cannot make an application execute concurrently when it was not designed to do so. In addition to communication, the J-Orchestra applet runtime provides various convenience services such as access to the properties of the server JVM, a capacity for terminating the server process, and a facility for browsing the server's file system efficiently.

Chapter V presents several case studies of successfully appletizing realistic, third-party applications, confirming the benefits of the approach. Specifically, our measurements show that the appletized applications perform favorably both in terms of network traffic and overall responsiveness compared to using a remote X display for controlling and monitoring the applications. Taking these results into account, it is safe to say that appletizing,

having the benefits of automation, flexibility, ease of deployment, and good performance, can be a useful tool for software evolution.

## 4.8 Run-Time Performance

This section examines issues of run-time performance of programs partitioned with J-Orchestra. First of all, it is important to state that the performance characteristics of a partitioned application depend primarily on the ability to derive a good placement for anchored groups and to determine performance-improving object mobility scenarios. In that respect, it is the user who has to estimate the potential data exchange patterns between network sites, possibly assisted by the J-Orchestra profiling tool. Thus this section is not concerned with estimating overheads caused by bad partitioning decisions. Rather it looks at the indirection overheads specific to the J-Orchestra rewrite. Although anchoring by choice can practically eliminate the indirection overheads of the J-Orchestra rewrite, it is worth examining how high these overheads can be in the worst case. Section 4.8.1 presents measurements of these overheads and details the local-only optimization employed in J-Orchestra.

### 4.8.1 Indirection Overheads and Optimization

#### 4.8.1.1 Indirection Overheads

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 [2]) 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. Figure 4-4 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 4-4.** J-Orchestra worst-case 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 Figure 4-5 later). Although a 37% slowdown of local processing can be acceptable for some applications, for many others it

is too high. Recall, however, that this would be the overhead of the J-Orchestra rewrite for a partitioning where all application objects were mobile. Anchoring by choice all but a few mobile classes completely eliminates this overhead.

#### 4.8.1.2 Local-Only Optimization

Recall that remote objects extend the RMI class `UnicastRemoteObject` to enable remote execution. The constructor of `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 *local-only* version of the remote object is created directly. A local-only object is isomorphic to a remote one but with a different name and without inheriting from `UnicastRemoteObject`. A proxy continues to point to such a local-only object until the application attempts to use the proxy in a remote method call. In that case, the proxy converts its local-only object to a remote one using a special conversion constructor. This constructor reassigns every member field from the local-only 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 (even such recent standards as .NET Remoting) 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 local-only 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 (Figure 4-5). 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 4-5.** Effect of lazy remote object creation (local-only objects) and J-Orchestra indirection

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

## 4.9 Java Language Features And Limitations

J-Orchestra needs to handle many Java language features specially in order to enable partitioning of unsuspecting applications. Features with special handling include inheritance, static methods and fields, object creation, arrays, object identity, synchronization, reflection, method access modifiers, garbage collection, and inner classes. We do not

describe the low-level specifics of dealing with every language feature here, as they are mostly straightforward—the interested reader should consult this publication [87] for more details. Nevertheless, it is interesting to survey some of the limitations of the system, both in its safety guarantees and in offering a complete emulation of a single Java VM over a distributed environment.

### 4.9.1 Unsafety

As mentioned in Section 4.4, there will always be unsafeties in the J-Orchestra classification, but these are inherent in the domain of automatic partitioning and not specific to J-Orchestra. No partitioning algorithm will ever be safe without assumptions about (or analysis of) the platform-specific binary code in the system classes. System code can always behave badly, keeping aliases to any object that gets created and accessing its fields directly, so that no proxy can be used instead. Additionally, several objects are only created and used implicitly by native code, without their presence ever becoming explicit at the level of the interface between system and application code. For example, every site is implicitly associated with at least one thread object. If the application semantics is sensitive to all threads being created on the same machine, then the execution of the partitioned application will not be identical to the original one. Similarly, every JVM offers predefined objects like `System.in`, `System.out`, `System.err`, `System.properties` and `System.exit`. The behavior of an application using these stateful implicit objects will not be the same on a single JVM and on multiple ones. Indeed, it is not even clear that there is a single correct behavior for the partitioned application—different policies may be appropriate for different scenarios. For example, when one 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.

### 4.9.2 Conservative classification

The J-Orchestra classification is quite conservative. For instance, it is perfectly reasonable to want to partition an application so that two different sites manipulate instances of a certain anchored unmodifiable class. For example, two different machines may need to use graphical windows, but without the windows manipulated by code on one machine ever leaking to code on the other. J-Orchestra cannot tell this automatically since it has to assume the worst about all references that potentially leak to native code. Thus, partitionings that require objects of the same anchored unmodifiable class to be created on two different sites are not safe according to the J-Orchestra classification. This is a problem that is commonly encountered in practice. In those cases, the user needs to manually override the J-Orchestra classification and assert that the classes can safely exist on two sites. Everything else proceeds as usual: the translation wrapping/unwrapping technique is still necessary, as it enables indirect access to anchored unmodifiable objects (e.g., so that code on site *A* can draw on a window of site *B*, as long as it never passes the remote reference to local unmodifiable code).

### 4.9.3 Reflection and dynamic loading

Reflection can 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. 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. 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 at load time. The J-Orchestra classification is a whole-program analysis and cannot be done incrementally: unmodifiable classes may be loaded and anchored on some nodes before loading another class makes apparent that the previous anchorings are inconsistent.

### 4.9.4 Inherited limitations

J-Orchestra inherits some limitations from its underlying middleware—Java RMI. These limitations are better addressed uniformly at the middleware level than by J-Orchestra. One limitation has to do with efficiency. Although RMI efficiency has improved in JDK 1.4, RMI still remains a heavyweight protocol. Another limitation concerns distributed garbage collection. 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.

Additionally, J-Orchestra does not explicitly address security and administrative domain issues—indeed the J-Orchestra rewrite even weakens the protections of some

methods, e.g., to make them accessible through an interface. We assume that the user has taken care of security concerns using an orthogonal approach to establish a trusted domain (e.g., a VPN).

## 4.10 Conclusions

Accessing remote resources has now become one of the primary motivations for distribution. In this chapter we have shown how J-Orchestra allows the partitioning of programs onto multiple machines without programming. Although J-Orchestra allows programmatic control of crucial distribution aspects (e.g., handling errors related to distribution) it neither attempts to change nor facilitates changing the structure of the original application. Thus, J-Orchestra is applicable in cases in which the original application has loosely coupled parts, as is commonly the case of controlling multiple resources.

Although J-Orchestra is certainly not a "naive end-user" tool, it is also not a "distributed systems guru" tool. Its ideal user is the system administrator or third-party programmer who wants to change the code and data locations of an existing application with only a superficial understanding of the inner workings of the application.

We believe that J-Orchestra is a versatile tool that offers practical value and interesting design ideas. J-Orchestra is interesting on the technical front as the first representative of partitioning tools with what we consider important characteristics:

- use of a high-level language runtime, such as the Java VM or the Microsoft CLR; performing modifications directly at the binary level.
- no changes to the runtime required for partitioning.

- provisions for correct execution even in the presence of code unaware of the distribution (e.g., Java system code).

While this chapter has concentrated on the motivation, design, and implementation issues of J-Orchestra, Chapter V looks at the applicability of the automatic partitioning approach and presents several case studies of successfully partitioning various applications.