J-Orchestra: Enhancing Java Programs with Distribution Capabilities

ELI TILEVICH
Virginia Tech
and
YANNIS SMARAGDAKIS
University of Massachusetts

J-Orchestra is a system that enhances centralized Java programs with distribution capabilities. Operating at the bytecode level, J-Orchestra transforms a centralized Java program (i.e., running on a single Java Virtual Machine (JVM)) into a distributed one (i.e., running across multiple JVMs). This transformation effectively separates distribution concerns from the core functionality of a program. J-Orchestra follows a semi-automatic transformation process. Through a GUI, the user selects program elements (at class granularity) and assigns them to network locations. Based on the user's input, the J-Orchestra backend *automatically partitions* the program through compiler-level techniques, without changes to the JVM or to the Java Runtime Environment (JRE) classes. By means of bytecode engineering and code generation, J-Orchestra substitutes method calls with remote method calls, direct object references with proxy references, etc. It also translates Java language features (e.g., static methods and fields, inheritance, inner classes, new object construction, etc.) for efficient distributed execution.

We detail the main technical issues that J-Orchestra addresses, including its mechanism for program transformation in the presence of unmodifiable code (e.g., in JRE classes) and the translation of concurrency and synchronization constructs to work correctly over the network. We further discuss a case study of transforming a large, commercial, third-party application for efficient execution in a client server environment and outline the architectural characteristics of centralized programs that are amenable to automated distribution with J-Orchestra.

Various aspects of the work described here were discussed in prior publications:
Liogkas, N., MacIntyre, B., Mynatt, E.D., Smaragdakis, Y., Tilevich, E., Voida, S. Automatic partitioning: Prototyping ubiquitous-computing applications. IEEE Pervasive Computing 03,3,40–47. © 2003 IEEE. With kind permission of IEEE.
Tilevich, E., Smaragdakis, Y., and Handte, M. Appletizing: Running legacy Java code remotely from a Web browser. In the IEEE International Conference on Software Maintenance (ICSM 2005). © 2003 IEEE. With kind permission of IEEE.
Tilevich, E. and Smaragdakis, Y., J-Orchestra: Automatic Java application partitioning. In Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, LNCS 2374, 178-204. © Springer-Verlag Berlin Heidelberg 2002. With kind permission of Springer Science+Business Media.
Tilevich, E. and Smaragdakis, Y., Portable and efficient distributed threads for Java. In Proceedings of the ACM/IFIP/USENIX Middleware Conference. Springer-Verlag, LNCS 3231, 478-492. © IFIP International Federation for Information Processing 2004. With kind permission of Springer Science+Business Media.
The authors gratefully acknowledge support from the NSF under Grants No. CCR-0220248 and CCR-0238289.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Enhancement*; D.1.2 [**Programming Techniques**]: Automatic Programming—*Program transformation*; D.1.2 [**Programming Techniques**]: Automatic Programming—*Program synthesis*

General Terms: Experimentation, Languages

Additional Key Words and Phrases: Separation of concerns, Distributed Computing, Java, Middleware, RMI, Bytecode engineering

1. INTRODUCTION

Separation of concerns is the holy grail of computing. The term refers to dividing a computing problem into parts so that different facets are isolated and reasoning can be performed independently. Separation of concerns has been a guiding principle for controlling the complexity of software ever since Dijkstra [1982] coined the term in 1974, some thirty five years ago. The question of which concerns can be effectively and efficiently separated from the core functionality of a computer program has been a central one for multiple computer science sub-disciplines.

Clear cut answers to this question identify the concerns that either can be effectively separated or are not amenable to separate treatment. Such cross-cutting concerns as *logging* and *persistence* [Atkinson et al. 1996] lend themselves to natural separation, whereas *efficient parallelism* (i.e., an efficient parallel solution for a sequential algorithm), *transactions*, and *failure handling* [Kienzle and Guerraoui 2002] do not.

The question of whether distribution can (or even should) be introduced transparently to an unaware centralized program is still under debate. For example, Waldo et al.'s well-known "Note on Distributed Computing" [Waldo et al. 1994] argues that "papering over the network" is ill-advised due to the differences in performance, different calling semantics, and the possibility of partial failure. (Other reasons such as direct memory access do not apply to the language environments of Java and C#.) On the other hand, multiple successful, widely deployed distributed systems (e.g., the NFS distributed file system) owe their success to hiding the network from unsuspecting applications. At the level of the programming model, distributed shared memory (DSM) systems [Yu and Cox 1997; Bal et al. 1998; Aridor et al. 1999] aim at hiding the differences between the centralized and distributed execution models from the programmer.

The J-Orchestra project contributes to this ongoing discussion by examining the issues of enhancing Java programs with distribution capabilities. The aim of the J-Orchestra project has been to provide novel software tools for distributed computing that are more convenient to use from the programmer's perspective (i.e., closer to the familiar centralized programming model). At the same time, the J-Orchestra users remain cognizant of the differences between the centralized and distributed execution models.

J-Orchestra is a system for *automatic application partitioning*—the process of enhancing an existing centralized program with distribution capabilities, without needing to rewrite its source code or needing to modify existing runtime systems.

An automatic partitioning system allows users to describe the location of system resources, and performs a rewrite of the program binaries to introduce appropriate distribution mechanisms. This technique contrasts sharply with traditional distribution middleware such as Java RMI [Wollrath et al. 1996], which automates the mechanics of distributed communication, but requires the application designer to explicitly encode decisions about the distribution structure of the application in the source code itself.

Automatic application partitioning is a refinement of the Distributed Shared Memory approach with one important element: Automatic partitioning aims to add distributed execution capabilities by transforming the program, and not by using a specialized runtime system. That is, automatic partitioning attempts to imitate many changes that a human programmer would have to perform by hand, with the partitioning tool handling the majority of distribution issues automatically.

By making distribution transparent and not requiring changes to the runtime system, automatic application partitioning provides ease-of-deployment advantages compared to the distribution technologies that require custom runtimes. It is easier to deploy an application with a standard Java VM (which supports a variety of third party libraries and can often be found precompiled for handheld devices such as PDAs and cell phones) than with a specialized VM that supports distribution.

The general structure of the process is simple: the partitioning tool takes as input a regular program and user-supplied location information for the program's code/data (both represented by program classes) and external resources (e.g., images, localization resources, etc.); the tool rewrites the program so that the code and data divide into parts that can run in the desired locations. Any communication between parts of the program at different locations automatically becomes remote.

This simple idea produces significant complications for realistic programs with data shared through pointers. Because centralized programs assume a single, shared address space, the same abstraction must be maintained over a network. Data shared through pointers in the centralized version must continue to be shared in the distributed program. Many pointers, or "references" in Java, must be transformed into indirect references (that is, references to a "proxy" object) that could point either to local objects or to objects over the network. In addition, other transformations must take place including:

—replacing direct field accesses to other objects with method calls

—replacing constructor calls with calls to factory methods

—replacing references to objects with references to special remotely-accessible wrapper objects

—transmitting synchronization requests over the network while maintaining thread identity over remote calls

To complicate matters further, objects from transformed code may have to interface with code that cannot be modified. For example, a data type representing a disk file could be accessed by code inside the runtime system as well as by application code: The application code has to see the transformed version of the object, as it may need to access it remotely. The VM code cannot be modified, however,

and expects the original form of the file object. Not surprisingly, much of the complexity associated with automatic partitioning systems is related to dealing with unmodifiable code.

J-Orchestra is a state-of-the-art automatic partitioning system. It is GUI-based, works on Java, and performs all transformations at the bytecode level. The J-Orchestra user sees a view of all the classes, both application-level classes and Java system classes involved in an application. The user's input consists of assigning groups of classes to network sites. The system then rewrites the application code to effect the partitioning—the resulting program uses standard middleware (Java RMI) for communication. J-Orchestra's partitioning does not need to modify either the JVM or its runtime classes.

At the technical level, the main contributions of J-Orchestra concern its handling of unmodifiable code, and of various Java features—notably, thread synchronization. These are responsible for the system's scalability. J-Orchestra has been used to successfully partition unsuspecting, third-party, binary-only applications of substantial size. Prior systems offer no similar validation, mainly due to limitations in handling unmodifiable code. For instance, the Addistant tool [Tatsubori et al. 2001] shifts onto the user the burden of ensuring that objects transformed for distribution are never accessed by unmodified code inside the VM—a task requiring full knowledge of program behavior, which is prohibitive for large, third-party programs.

This article is effectively a retrospective of the entire J-Orchestra project, various aspects of which were described in previous publications [Tilevich and Smaragdakis 2002b; 2002a; 2004; Liogkas et al. 2004; Tilevich et al. 2005; Tilevich and Smaragdakis 2006]. Thus, the article collects and updates our J-Orchestra experiences, with the benefit of a complete picture. Perhaps the greatest advantage of this hindsight is an understanding of which are the important elements of the J-Orchestra project, which the article attempts to emphasize.

We next describe J-Orchestra, as well as the insights gained from it. Section 2 introduces J-Orchestra via a case study of partitioning a large, centralized, third-party application for efficient execution in a client server environment. Section 3 describes the main facets of the J-Orchestra translation process. Section 4 details how J-Orchestra translates various Java language features. Section 5 presents the J-Orchestra approach to maintaining Java concurrency and monitor style synchronization constructs across the network. Section 6 gives a retrospective summary of the lessons learned. Section 7 compares J-Orchestra to other research aimed at facilitating distributed application development, and Section 8 presents concluding remarks.

2. WHY PARTITION PROGRAMS: THE J-BITS CASE STUDY

Before introducing automatic application partitioning, one must ask what benefits could be gained from introducing distribution to a centralized program. The foremost reason for distributing a program with J-Orchestra is to take advantage of remote hardware or software resources (e.g., a processor, a database, a graphical screen, or a sound card). Several special-purpose technologies do this already: distributed file systems allow storage on remote disks; Java applets move graphics-

producing code from a server to a client with the screen on which the graphics will be displayed; and network printer protocols let users print remotely.

The distribution decisions of these special-purpose distribution technologies, however, are hard-coded. For example, Java applets require that the entire program move across the network. The advantage of automatic partitioning is that it provides greater flexibility. By splitting a program into arbitrarily-shaped partitions, automatic partitioning can create distributed applications that are optimized for different networking environments. For instance, if a graphical representation can be computed from less data than it takes to transfer the entire graphical representation over the network, such distribution would have an advantage.

With the above observations in mind, we next showcase J-Orchestra through a case study of partitioning JBits, a large, centralized, third-party application for efficient execution in a client server environment. JBits is an FPGA graphical monitor and simulator by Xilinx [Guccione et al. 1999]—a web search shows many instances of industrial use. The JBits GUI is rich with multiple graphical areas presenting the state of the real or simulated hardware. The GUI allows connecting to various hardware boards and simulators and depicting them in graphical form. It also allows stepping through an execution, offering multiple logical views of a hardware board, each of which can be zoomed in and out, scrolled, and so forth.

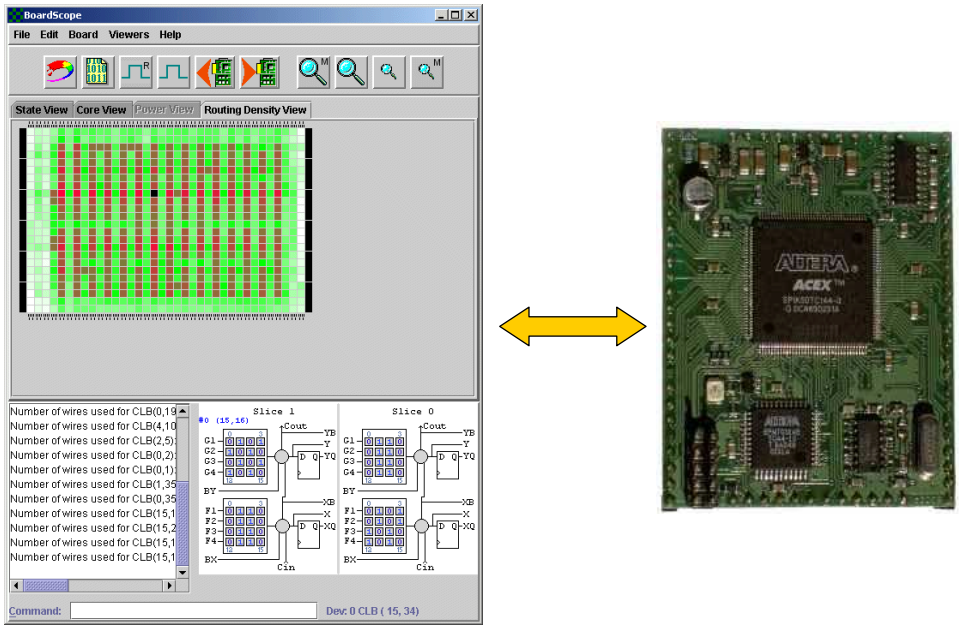

Fig. 1. JBits Usage Scenario.

A typical deployment environment for JBits is a workstation connected to a hardware board (Figure 1). In other words, the deployment environment utilizes two main hardware resources: a graphical screen (to render the application GUI) and a physical connection to an external device (the board).

A user of JBits may wish to access the application over a network connection—e.g., over a home DSL link.[1] The latency/bandwidth ratio of this network connection is likely to render remote desktop applications (e.g., VNC [Richardson et al. 1998], X [Scheifler 1987]) inadequate for this task. Because these technologies transfer the graphical data representing a computer screen over the network, the amount of data transferred for a complex GUI can be significant, resulting in higher latency and poor user experience. In contrast, automatic partitioning can split a program so that the graphics rendering code is entirely on the client machine. This way, only the simulation data could be transferred over the network, with most graphical operations requiring no network traffic.

JBits was given to us in bytecode-only form. The installed distribution (with only Java binary code counted) consists of 1,920 application classes that have a combined size of 7,577 KBytes. These application classes also use a large part of the Java system libraries. We have no understanding of the internals of JBits, and only limited understanding of its user-level functionality. For our partitioning, the vast majority (about 1,800) of the application's classes are placed on the server. As we will see in Section 3, objects of permanently co-located classes access each other directly and impose no overhead on the application's execution. This is particularly important in this case, as the main functionality of JBits is the simulation, which is compute-intensive. 259 classes are placed on the client (i.e., GUI) site. Of these, 144 are JBits application classes and the rest are classes from the Java system's graphical packages (AWT and Swing). (We later discuss a variation in which we make some objects mobile.)

In our measurements, we compare the partitioned application's behavior to using a remote X display [Scheifler and Gettys 1986] to remotely control and monitor the application. Since JBits is an interactive application and we could not modify what it does, we got measurements of the data transferred and not the time taken to update the screen (i.e., we measured bandwidth consumption but not latency). Our experience is that partitioning is an even greater win in terms of perceived latency: In all cases, the overall responsiveness of the partitioned versions is much better than using remote X displays. This is hardly surprising, as many GUI operations require no network transfer. Note that the data transfer numbers do not depend on the machines or network used. For reference, however, most of our experimentation was over a "slow" 10 Mbps ethernet link.

The partitioned JBits can be made to perform arbitrarily better than a remote X-Window display. For instance:

—JBits has multiple views of the simulation results ("State View", "Power View", "Core View", and "Routing Density View"). Switching between views is a completely local operation in the J-Orchestra partitioned version: no network transfers are caused. In contrast, the X window system needs to constantly refresh the graphics on screen. For cycling through all four views, X needed 3.4 MBytes transferred over the network.

[1]This scenario is not just realistic, but real. We were introduced to JBits because a computer architecture researcher (and former colleague) expressed excitement in the J-Orchestra project from its very early days, because it would allow him to use JBits from home.

—JBits has deep drop-down menus (e.g., a 4-level deep menu under "Board→Connect"). Navigating these drop-down menus is a local operation for the J-Orchestra partitioned application, but not for remote access with the X window system. For interactively navigating 4 levels of drop-down menus, X transferred 1.8 MBytes of data.

—GUI operations like resizing the virtual display, scrolling the simulated board, or zooming in and out (four of the ten buttons on the JBits main toolbar are for resizing operations) do not result in network traffic with the partitioned JBits. In contrast, the remote X display produces heavy network traffic for such operations. With our example board, one action each of zooming-in completely and zooming-out results in 3.5 MBytes of data transferred. Scrolling left once and down once produces about 2 MBytes of data over the network with X, but no network traffic with the J-Orchestra partitioned version. Continuous scrolling over a 10 Mbps link is unusably slow with the X window system. Clearly, a slower connection (e.g., DSL) is not suitable for remote interactive use of JBits with X.

Even for a regular board redraw, in which the partitioned JBits needs to transfer data over the network, less data get transferred than in the X version. Specifically, the partitioned version needs to transfer about 1.28 MB of data for a complete simulation step including a redraw of the view. The X window system transfers about 1.68 MBytes for the same task. Furthermore, J-Orchestra transfers these data using five times fewer total TCP segments, suggesting that, for a network in which latency is the bottleneck, X would be even less efficient. Although there may be ways (e.g., compression, or a more efficient protocol) to reduce the amount of data transferred by X, the important point is that some data transfer needs to take place anyway. In contrast, the partitioned version only needs to transfer a data object to the remote site, and all GUI operations presenting the same data can then be performed locally. For the cases that do produce network traffic, the partitioned version can also have its bandwidth requirements optimized by using a version of Java RMI with compression.

Experiment: Mobility. One of the capabilities of J-Orchestra is to allow objects to move from host to host, to exploit locality. Our above analysis did not include any such mobile objects. Even though many of the JBits objects could be made mobile, very few of them actually need to move in an interesting way. The one exception is JBits View Adaptor objects (instances of four classes with the `ViewAdaptor` suffix). View adaptors seem to be logical representations of visual components and they also handle different kinds of user events such as mouse movements. During our profiling we noticed that such objects are used both on the server and the client partition and in fact can be seen as carriers of data between the two partitions. Thus, no static placement of all view adaptor objects is optimal—the objects need to move to exploit locality. We specified a mobility policy that originally creates view adaptors on the client site, moves them to the server site when they need to be updated, and then moves them back to the client site. Surprisingly, object mobility results in more data transferred over the network. With mobile view adaptor objects and an otherwise indistinguishable partitioning, J-Orchestra transferred more than 2.59 MBytes per simulation step (as opposed to 1.28 MBytes without a mobility

policy). The reason is that the mobile objects are quite large (in the order of 300-400 KBytes) but only a small part of their data are read/written. In terms of bytes transferred it would make sense to leave these objects on one site and send them their method parameters remotely. Nevertheless, mobility results in a decrease in the total number of remote calls: 386 remote calls take place instead of 484 for a static partitioning, in order to start JBits, load a file and perform 5 simulation steps. Thus, the partitioned version of JBits with mobile objects may perform better for high bandwidth networks, in which latency is the bottleneck.

Discussion. Automatic partitioning is not free of limitations. Applications can be arbitrarily complex and can defy correct partitioning. More common in practice, however, is the case of applications that can be correctly partitioned (i.e., they do not employ unsupported Java features such as dynamic loading) yet require manual intervention to override conservative decisions of the J-Orchestra heuristic analyses. Partitioning JBits required some intervention (but no explicit programming) to arrive at a good partitioning within 1-2 days. For example, knowing only the JBits execution from the user perspective, we speculated that the integer arrays transferred from the server towards the GUI part of JBits could safely be copied to the client rather than accessed through a remote proxy. These arrays turned out to never be modified at the GUI part of the application. A more conservative rewrite would have introduced a substantial overhead to all array operations. Even in the less automatic cases, however, the expertise required to partition an application is analogous to that of a system administrator, rather than that of a distributed systems programmer. In the JBits case, we partitioned a 7.5 MB binary application without knowledge of its internals. Even though the partitioning was not automatic, the effort expended was certainly much less than that of a developer who would need to change an application with about 2,000 classes, more than 200 of which need to be modified to be accessed remotely.

Additional Partitioning Examples. We have used J-Orchestra to partition multiple Java programs and deploy them on diverse platforms. A common case is that of taking a straightforward centralized Java program, partitioning it, and deploying it on a mobile device that communicates with a central server or a laptop machine. One example is an application in which GUI actions on a PDA produce synthesized speech on a server machine. Another example is a smart controller for PowerPoint presentations. We started with a small Java application that controls PowerPoint through its COM interface. J-Orchestra was able to partition this application into a GUI and a back-end part, so that the GUI runs on a Linux PDA equipped with a wireless card and uses it to control PowerPoint running on a Windows laptop.

3. TRANSLATION PROCESS

Having outlined the user argument for our approach, we now describe the main design and implementation decisions in the J-Orchestra translation process.

3.1 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 runtime system while supporting unaware applications sets J-Orchestra apart from traditional Distributed Shared Memory (DSM) systems. (The related work section 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,[2] the lack of data sharing has been a fundamental assumption for some past work in partitioning systems—e.g., the Coign approach [Hunt and Scott 1999].

It is worth asking why mature partitioning systems have not been implemented in the past. For example, why no existing technology can partition a platform-specific binary (e.g., an x86 executable) to have different parts of the code 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 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. 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. Addressing the problem of adding indirection in the presence of unmodifiable code is a novel contribution of J-Orchestra.

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 [Shapiro 1986]. This creates an abstraction of shared memory in which proxies hide the 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

[2]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.

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 should be visible to the other, as if on a single machine. Figure 2 shows schematically the effects of the indirect referencing approach. This indirect referencing approach has been used in several prior systems [Philippsen and Zenger 1997; Spiegel 2002; Tatsubori et al. 2001].

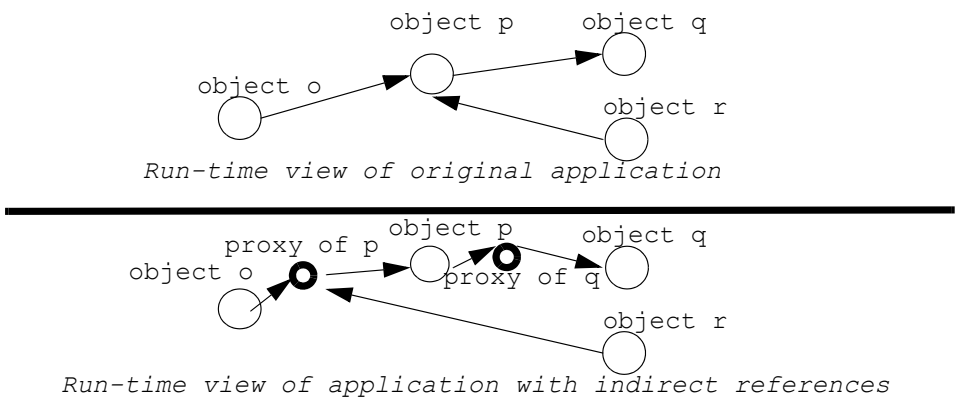

Fig. 2. Results of the indirect reference approach schematically. Proxy objects could point to their targets either locally or over the network.

Since one of the requirements is to leave the runtime system unchanged, J-Orchestra cannot change the JVM's pointer/reference abstraction. Instead, it rewrites the entire partitioned application to introduce proxies for every reference in the application.[3] 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 requires a global rewrite for correctness, yet some code cannot be modified, as it is part of the runtime system. In our context (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

[3]More precisely, it is acceptable to think that every reference in the transformed application becomes a proxy reference *when discussing correctness/semantics*. This is not accurate in practice, as it would introduce significant overhead. Only a small number of references end up being through proxies, but this optimization can be ignored at first approximation.

files: the file operation code is part of the Java VM (i.e., in machine-specific binary code). 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. (For simplicity, we assume that only system classes contain native code—i.e., the application code itself is "pure Java".)

J-Orchestra effectively solves many of the problems of dealing with unmodifiable code by partitioning *around* unmodifiable code (i.e., by computing which data are *not* affected by unmodifiable code and enabling full partitioning for them, while allowing only remote access for data that are manipulated by 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. We discuss this and other specifics of the J-Orchestra program transformations next.

3.2 Before Rewriting: Classification

J-Orchestra classifies classes into two categories, with each class's category determining the set of transformations it goes through during rewriting. The user interaction with J-Orchestra is through a graphical interface. Once the user imports application classes, J-Orchestra determines the full set of system classes they access, and runs a preliminary classification algorithm, based on the use of native code. The user can override any classification decisions. The result of classification is to split classes into two groups: *anchored* and *mobile*.

—**Anchored classes** (after preliminary classification) either contain native code or can have their instances accessed inside native code. Intuitively, 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. Classes within the same anchored group reference each other directly (i.e., the same native code has access to objects of both classes) 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. For instance, all classes accessing a single hardware resource (such as GUI classes, or classes using the Java Speech API) will likely be clustered in the same group. In this way, the preliminary classification guarantees safety (under heuristic assumptions described in Section 4). The user can override the preliminary classification and break up an anchored group, if this is safe.
While classes within the same anchored group cannot be separated, different anchored groups can be placed on different network sites.

—**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*, overriding the system's preliminary classification.

Once the classification is finalized, the J-Orchestra GUI allows the user to assign network sites to anchored class groups, or individual mobile classes. For an anchored class group, this assigns the group's 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. In the end, J-Orchestra rewrites the original application and puts all the modified classes, generated supporting classes, and J-Orchestra run-time configuration files into `jar` files, one per destination network site. At run-time, the application is deployed with a small (pure Java) J-Orchestra runtime library, which handles remote object creation, object mobility, and various bookkeeping tasks.

3.3 Rewriting Engine

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,[4] 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 elements of the rewriting approach.

3.3.1 *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. For instance, consider an original class `A`:

```
class A {
    void foo () { ... }
}
```

`A`'s proxy will be generated in source code form (slightly simplified) as follows:

```
class A implements java.io.Externalizable {
    //ref can point either to remote implementation class or RMI stub.
    A__interface ref;
    ...
    void foo () {
        try {
            ref.foo ();
        } catch (RemoteException e) {
            //let the user provide custom failure handling
        }
    }//foo
}//A
```

[4]We use the BCEL library [Dahm 1999] for bytecode engineering.

That is, a proxy class has the same method interface as the original class and dynamically delegates to a *remote implementation* object. Remote implementation classes extend the RMI class `UnicastRemoteObject`. This means that they can be registered as Java RMI remote objects and 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. 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. For our example class `A`, its remote interface and its remote implementation class will have the form:

```
//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 {
       ... // modified body of original A.foo()
    }
}
```

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 use standard Java RMI facilities to control distributed execution at appropriate times—e.g., 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.

Objects of anchored classes, on the other hand, are accessed by other objects on the same site *without* any proxy indirection. (Recall that classes become anchored either by necessity, when the J-Orchestra classification determines that their objects can be accessed by native code of other classes in the anchored group, or by choice, when the user is trying to avoid the overhead of proxy indirection for accesses on a certain site.) Yet anchored classes still require proxies, so that their methods can be called by other parts of the application (i.e., mobile objects, or remote code). Such proxies provide similar functionality as for mobile classes, but do not assume the names of their original classes. 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, and to avoid the need to modify system-level anchored classes. (The Java VM places constraints on adding, modifying, or replacing system packages.) Regular proxies, as well as remote implementation versions are created for translators, exactly like for mobile classes. Proxy objects

could point to their targets either locally or over the network. Figure 3 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.

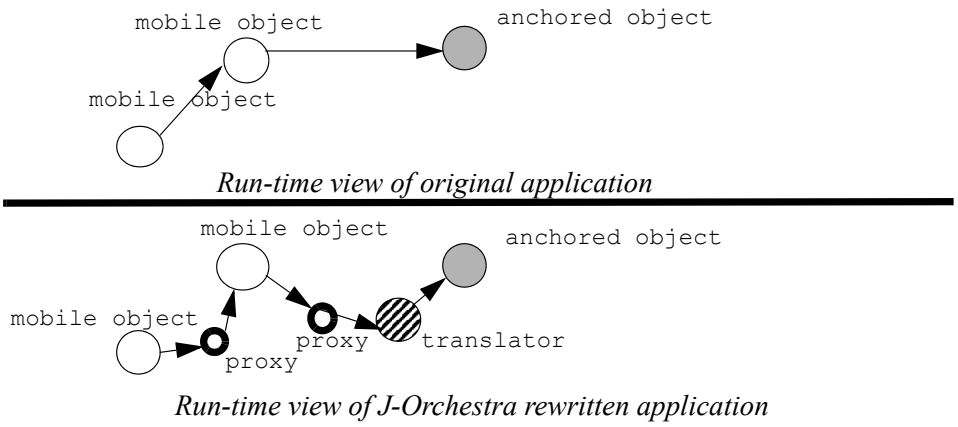

Fig. 3. Results of the J-Orchestra rewrite schematically, for both mobile and anchored objects. Proxy objects could point to their targets either locally or over the network.

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. For instance, in Figure 4, a mobile application object `o` holds a reference `p` to an object of (anchored) type `java.awt.Point`. Object `o` can pass reference `p` as an 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 (e.g., has a native code implementation). Thus, the unmodifiable code of method `contains` 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 other systems with a rewrite strategy similar to that of J-Orchestra [Philippsen and Zenger 1997; Spiegel 2002; Tatsubori et al. 2001] do not offer a direct-indirect reference translation mechanism. Thus, in those systems the partitioned application is correct only if objects passed to unmodifiable (system) code are guaranteed to remain on the same site as that code. In contrast, J-Orchestra ensures the absence of such run-time errors, by combining classification information with the dynamic reference translation mechanism: No object is

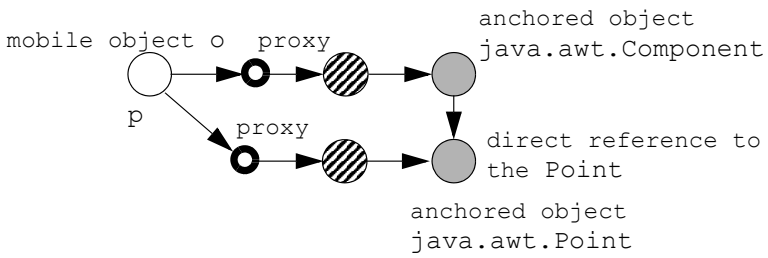

Fig. 4. Mobile code refers to anchored objects indirectly (through proxies) but anchored code refers to the same objects directly. Each kind of reference should be derivable from the other.

accessed directly if it could be remote.

Reference 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) one 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 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());
   }
}
```

Note that the classification algorithm is precisely what ensures that communication between anchored code and mobile code is only done through method calls

in translator classes.[5] As we will see later, other data exchange would result in placing the relevant types in the same anchored group.

3.4 Object Mobility Specifics

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.

Object mobility enables a per-instance rather than per-class distribution policy: two objects of the same 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 described in the mobile objects literature [Black et al. 2007]. In terms of mechanisms, our implementation bears many similarities to the one in JavaParty [Philippsen and Zenger 1997].

Specifically, J-Orchestra supports mobility through a programming interface and runtime services. J-Orchestra proxies are generated in source code form, which makes it fairly straightforward to generate additional mobility-specific methods in mobile classes' proxies. The programmer can then use these generated methods as primitives for implementing 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. For example, the code below demonstrates how the J-Orchestra mobility API can be used to specify that the parameter `p` of the remote method `foo` be moved to the side of the remote method invocation.

```
//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.
```

[5]This is only true at a first approximation, if we ignore anchoring by choice. Anchoring classes by choice introduces more complications, as described in [Tilevich 2005, Ch.4.5.2]. The result is that wrapping/unwrapping code occasionally needs to be added to call sites outside translator objects—i.e., the caller class itself needs to be modified. Nevertheless, this is just an engineering issue and the essence of the approach remains. A common case of such wrapping is self-reference, i.e., access to an object through the "this" reference. An object always refers to itself directly, but may pass a reference to other code that accesses the object indirectly, thus necessitating wrapping.

```
        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) {...}
}
```

In addition, the J-Orchestra mobility API contains methods that can be used to move "this" object (i.e., the object 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) {...}
```

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—e.g., "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.

4. HANDLING JAVA LANGUAGE FEATURES

In this section, we describe how J-Orchestra handles various features of the Java language. Some parts of the translation (e.g., the handling of static methods) are straightforward and only add engineering complexity. The handling of other elements (e.g., object identity, unmodifiable code), however, is far from trivial. We treat concurrency and synchronization separately in Section 5 as it is a complex and self-contained topic.

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

4.1 Classification of Unmodifiable Code

As described in Section 3, before the J-Orchestra rewrite can introduce indirection to enable a distributed memory abstraction, it must first determine which program classes can be safely indirected. This is the task of the J-Orchestra classifier, which categorizes each class as anchored or mobile and determines anchored class groups. (More precisely, this is the *preliminary* classification that J-Orchestra performs automatically, which the user can override for the final classification.)

Conceptually, the preliminary classification process has a simple goal. It needs to compute 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 if the unmodifiable code tries 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), a run-time error will occur. 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.

To obtain the classification information precisely, one would need to analyze the implementation of native code. It is highly complicated to obtain the source code for *all* platform specific runtimes, or to require VM implementors to export a model of the behavior of their system just for the purpose of enabling safe indirection. Thus, J-Orchestra follows a more pragmatic approach, using the *type information* at the native code interface to derive a native code behavior model. The base J-Orchestra rules for inferring anchored classes are as follows:

(1) A system class with native methods is anchored.
(2) A system class used as a parameter or return type for a method or static method in a anchored class is *co-anchored* with that class (i.e., anchored and in the same group).
(3) If a system class is anchored, then all class types of its fields or static fields are co-anchored.
(4) If a system class, other than `java.lang.Object`, is anchored, then its subclasses and superclasses are co-anchored.

(The rules reflect the essence of the analysis heuristic. We do not discuss variations such as arrays or exceptions—these are handled similarly to regular classes holding references to the array element type and method return types, respectively. Note that interface access does not impose restrictions since an interface cannot be used to directly access state.) Rule 1 exists because no indirection technique can guarantee to capture all field updates of an instance of a class with a native method. The native method can always perform field updates without any indirection. Rule 2 is justified with a similar argument: if an object can be passed to native code, native code can alias it and (either during the native method execution or during a later invocation) change its state. Furthermore, the rule can be applied transitively: if a class is anchored then we cannot replace all its uses with uses of an instrumented version in some application package. Then all objects used as arguments of any method (even non-native) may have their fields accessed directly. Rule 3 is like rule 2 but for fields: native code can access objects transitively reachable from an object that leaks to native code. Rule 4 is based on the specifics of the J-Orchestra proxy indirection scheme and the Java restrictions on changing system packages. (In other contexts, rules 2 and 4 can be weakened, as we explain elsewhere [Tilevich and Smaragdakis 2006].)

With these rules and the assumption that native code in different system libraries (e.g., graphics code and sound code) does not share state, J-Orchestra can successfully place different system classes on different machines, while allowing safe access. The safety is heuristic, however. It is impossible to recognize the interactions of system classes with native code without making assumptions regarding native code behavior. For instance, all classes in Java are subclasses of `java.lang.Object`, which does have native methods. In theory, any native method can be receiving an `Object`-typed argument, discovering its actual type using reflection and performing on the object some action (e.g., reading fields) that would be undetected by the J-Orchestra indirection. Thus, the correctness of the J-Orchestra classification is mainly based on two assumptions regarding system classes:

—System classes without native methods have no special semantics. (Native code

never treats their objects any differently from user-defined objects.)

—Native methods do not use dynamic type discovery (reflection, downcasting, or any low-level type information recovery) on objects supplied through method arguments.

The first assumption essentially states that the JVM is not allowed to handle different types of objects specially when the objects just use plain bytecode instructions. For instance, the JVM is not allowed to detect the construction of an object of a "special" type and keep a reference to this object that native code can later use for destructive state updates. This is a reasonable assumption, conforming to good software design practices. The second assumption states that native code is strongly typed: if a reference is declared to be of type `T`, it can never be used to access fields (method calls are fine) of a subclass of `T`. For instance, the assumption prohibits native methods from taking an `Object`-typed argument, checking if it is actually of a more specific type (e.g., `Thread` or `Window`), casting the object to that type and directly accessing fields or methods defined by the more specific type. This assumption also encodes a good design practice: code exploits the static type system as much as possible for correctness checking. Although the assumption may be violated locally, the hope is that it is rarely violated over the bytecode-native code boundary.

These assumptions generally hold true with only few exceptions. The first assumption does not hold, for instance, for classes in the `java.lang.ref` package. The second assumption does not hold in the implementation of reflection classes themselves. As we describe elsewhere [Tilevich and Smaragdakis 2006], we validated these assumptions with two experiments: a code inspection of the C/C++ native code of Sun's JDK 1.4.2 for Solaris, and a dynamic analysis. Our code inspection showed that the native implementations respect static typing and use dynamic type discovery (i.e., reflection) sparingly. For instance, we found only 69 instances of the JNI function `IsInstanceOf`, which is the main way to do dynamic type discovery in native code. For comparison, there were thousands of uses of the Java user-level counterpart, `instanceOf` in plain Java code in the Java system libraries. Our dynamic analysis examined executions of mainstream Java benchmarks (including the largest applications from the DaCapo suite and SPEC JVM'98) and found only two instances of native code accesses that were not predicted by J-Orchestra's type-based classification heuristic.

Overall, our experience of using J-Orchestra for several years confirms that its type-based analysis is useful and quite safe in practice. In the absence of complete information on the behavior of native code, this analysis is a clear win. The only general-purpose alternatives are to either not support indirection for any system classes, or to leave the user with no assistance in determining the correctness of applying indirection.

4.2 Static Methods and Fields

J-Orchestra has to handle remote execution of static methods. This also takes care of remote access to static fields: just like with member fields, J-Orchestra replaces all direct accesses to static fields of other classes with calls to accessor and mutator methods. For handling static methods, J-Orchestra creates static delegator

classes for every original class that has any static methods. Static delegators extend `java.rmi.server.UnicastRemoteObject` and define all the static methods declared in the original class.

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

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

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

4.3 Inheritance

Proxy classes, translator classes, and remote classes all mimic the inheritance/subtyping hierarchy of their corresponding original classes. Furthermore, one can think of the generation process for proxy and implementation (i.e., translator or remote) classes as a covariant function: a proxy of a subtype is a subtype of the supertype's proxy. Replacing direct references with references to proxies preserves the original subtyping semantics: a proxy can be used when a supertype instance is expected.

Proxy classes use delegation to invoke methods on implementation classes. The base proxy class declares the *delegatee* field, which is shared by all proxy classes in a hierarchy. Constructors of all (derived) proxy classes initialize the delegatee field to the respective (derived) implementation type. Then each proxy in the hierarchy casts the delegatee field to its respective implementation type when necessary for delegation.

4.4 Object Creation

Creating objects remotely is a necessary functionality for every distributed object system. J-Orchestra enables remote object creation by handling proxies' constructors differently from other methods. First, a proxy constructor calls a special-purpose do-nothing constructor in its super class to avoid the regular object creation sequence. A proxy constructor creates objects using the services of the object factory. J-Orchestra's object factory is an RMI service running on every network

node where the partitioned application operates. Every object factory is parameterized with configuration files specifying a symbolic location of every class in the application and the URLs of other object factories. Object factory clients determine object locations, handle remote object creations, and maintain various mappings between the created objects and their proxies. The following example shows a portion of the constructor code in a proxy class `A`. As can be seen, constructing the proxy causes the creation of the appropriate remote object through a factory.

```
public A () {
    //call super do-nothing constructor
    super ((BogusConstructorArg)null);
    //check if we are already initialized or are called from a subclass
    if ((null != _remoteRef) || (!getClass ().equals (A.class)))
        return;
    ...
    //Call ObjectFactory
    try {
        _remoteRef = (A) ObjectFactory.createObject("A");
    } catch (RemoteException e) { ... }
}
```

4.5 Arrays

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

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

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

It is worth noting that the same wrapping/unwrapping needs to be performed for multidimensional anchored arrays. For instance, if a two dimensional array of integers is anchored, then before each of its constituent arrays is retrieved, it needs to be wrapped in a front-end for one dimensional integer arrays.

Determining whether an array needs to be anchored or can be mobile is an interesting problem. Although arrays are implemented in native code, we can safely assume that this code does not capture system-specific state and never directly accesses fields of the arguments to "store" and "load" methods, as the code has no knowledge of the types of the array elements. Therefore, arrays can be made mobile (unless they themselves leak to native code—see next). Note that this means that an array of objects of class `C` can be mobile even when class `C` is anchored—`C` objects may cross the native code boundary, but as long as arrays of `C` objects do not cross it, these arrays can be made mobile.

Nevertheless, the usual type-based anchored/mobile classification mechanism of J-Orchestra can be too restrictive when applied to arrays. Recall that according to the J-Orchestra classification, if a reference to a certain type can leak to native code, then all references to this type are made anchored. The problem is that the J-Orchestra classification algorithm is type based and primitive array types are anonymous types. The same type, e.g., `int[]`, can be used for very different purposes, but the J-Orchestra classification heuristic is conservative for safety. For instance, any application that passes an integer array to an anchored system class will have to treat all its integer arrays (of the same or lower dimension) as anchored on the same site! This restriction may even hinder the ability to safely place different Java system classes on different network sites. If two entirely unconnected system packages both exchange arrays of integers with some application's code, then both packages have to be placed on the same machine, because of the possibility that they both refer to the same array. So far, this problem is addressed only by having the user override the preliminary classification for arrays. Note that the issue concerns the read-write use of arrays: if arrays are only written by application code and read by system code (or vice-versa), they can safely be made mobile. The latter is a common pattern for arrays shared between application and system code.

4.6 Object Identity

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

J-Orchestra employs a similar scheme to handle anchored objects' wrapping. When an object is unwrapped and re-wrapped, we should ensure that the identity of the proxy (the "wrap" object) is preserved. Consider an example method `returnMyArgument` in anchored class `A` that takes an argument of another anchored class `B`.

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

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

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

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

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

4.7 Reflection and Dynamic Class Loading

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

We should note that offering full support for correctness under reflection is possible. For example, it is possible to create a J-Orchestra–specific reflection library that will mimic the interface of the regular Java reflection routines but will take care to always hide proxies. All reflection questions on a proxy object will instead be handled by the remote object. With bytecode manipulation, we can replace all method calls to Java reflection functionality with method calls to the J-Orchestra–specific reflection library. We have considered this task to be too complex for the expected benefit.

Similar observations hold regarding dynamic class loading. J-Orchestra is meant

for use in cases in which 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 incrementally: unmodifiable classes may be loaded and anchored on some nodes before loading another class makes apparent that the previous anchorings are inconsistent. The only safe approach would be to make all dynamically loaded classes anchored on the same network site.

4.8 Garbage Collection

Distributed garbage collection is a tough problem [Jones and Lins 1996]. J-Orchestra relies on the RMI distributed reference counting mechanism for garbage collection. This means that cyclic garbage, in which the cycle traverses the network, will never be collected. Nevertheless, this aspect is orthogonal to the goal of J-Orchestra—the system just inherits the garbage collection facility of the underlying middleware.

4.9 Inner Classes

On the Java language level, inner classes have direct access to all member fields (including private and protected) of their enclosing classes. In order to enable this access, the Java compiler introduces *synthetic* methods that access and modify member fields of enclosing classes. Synthetic methods are not visible during Java source-to-bytecode compilation. Synthetic methods also need to be accessed through a proxy, however. (The code inside a synthetic proxy method accesses the synthetic method of its remote class.) This presents a problem for J-Orchestra: although most J-Orchestra transformations are done directly in bytecode, proxies are created in source code form (so that the user can manually edit them if needed for failure or mobility handling). As a result, no Java compiler would be able to successfully compile proxies with synthetic methods, since they call methods that do not appear in the source. Removing the synthetic attributes (part of the meta-data maintained in class binaries) from methods in remote classes eliminates the problem. The removal does not violate the Java security semantics because synthetic methods have no access restrictions to begin with.

4.10 `System.out`, `System.in`, `System.err`, `System.exit`, `System.properties`

The `java.lang.System` class provides access to standard input, standard output, and error output streams (exported as pre-defined objects), access to externally defined "properties", and a way to terminate the execution of the JVM. In a distributed environment, it is important to modify these facilities so that their behavior makes sense. Different policies may be appropriate for different applications. For example, when any of the partitions writes something to the standard output stream, should the results be visible only on the network site of the partition, all the network sites, or one specially designated network site that handles I/O? If one of the partitions makes a call to `System.exit`, should only the JVM that runs that

partition exit or should the request be applied to all the remaining network sites? J-Orchestra allows defining these policies on a per-application basis. For this purpose, J-Orchestra provides classes called `RemoteIn`, `RemoteOut`, `RemoteErr`, `RemoteExit`, and `RemoteProperties`, whose implementation determines the application-specific policy. For example, all references to `System.out` are replaced with `RemoteOut.out` in all the rewritten code. An implementation of `RemoteOut.out` can return a stream that redirects all the messages to a particular network site, for example.

5. HANDLING CONCURRENCY AND SYNCHRONIZATION

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.)

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. We next discuss the problem and how J-Orchestra solves it. The solution is of independent research interest, as it improves on prior approaches in the general middleware context.

5.1 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, synchronized methods and code blocks for mutual exclusion, and monitor methods `Object.wait`, `Object.notify`, and `Object.notifyAll` for managing state dependence. (An excellent reference for multithreading in Java is Lea's textbook [Lea 1997].)

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`) (see Figure 5). Such an object is remotely accessible through the `RI` interface. That is, if a client holds an interface reference `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 [Birrell and Nelson 1984]: the client is really holding an indirect reference to `obj`. Reference `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 propa-

gate 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 `ri` instead. Similar problems exist for all other monitor operations. For instance, RMI cannot be used to propagate monitor operations such as `Object.wait` and `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.

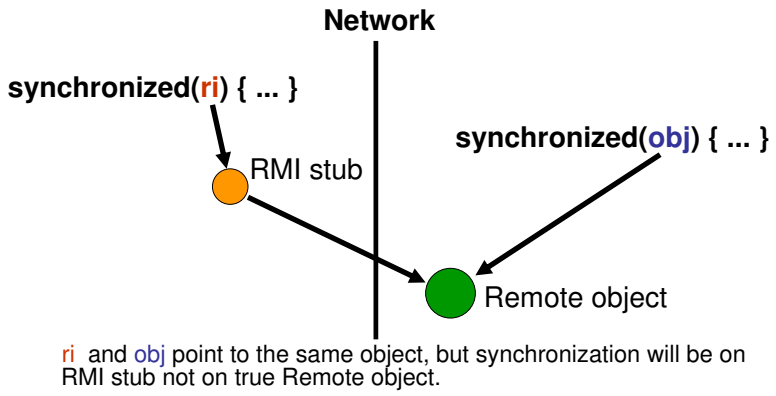

Fig. 5. Referencing a remote RMI object directly and over the network.

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 6 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.

5.2 Solution: Distribution-Aware Synchronization

The J-Orchestra approach to solving these distributed synchronization problems is twofold: First, we maintain per-site "thread id equivalence classes," which are

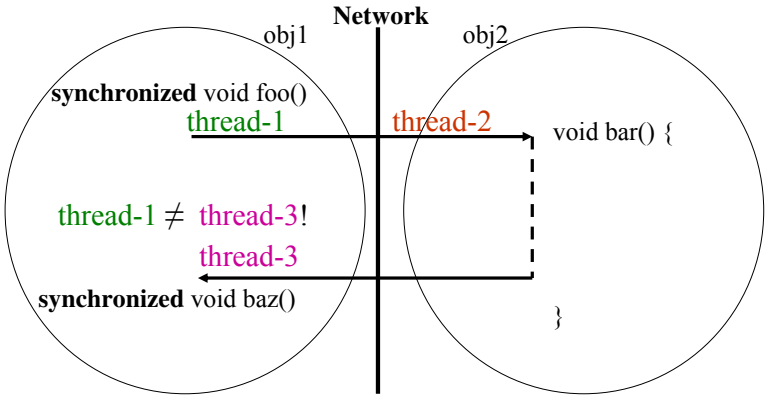

Fig. 6. The zigzag deadlock problem in Java RMI.

updated as execution crosses the network boundary. Second, we replace at the bytecode level all uses of standard synchronization constructs with method calls to a specialized per-site synchronization library. This synchronization library emulates the behavior of 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. If we know statically that the object is local, we avoid the runtime test and instead call a local synchronization operation. 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.

—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();
}
```

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. (Throughout the section, although all the changes are applied to the bytecode directly, we show source code for ease of exposition.)

```
//original code
void foo (Object some_remote_object) {
    jorchestra.runtime.diststhreads.wait_(this); //dispatched locally
    ...
    jorchestra.runtime.ThreadInfo.
        getThreadEqClass(some_remote_object).notify_();
    //get thread equivalence info from runtime and dispatch through RMI
}
```

(The last instruction is an interface call, which implies that each remote object needs to support monitor methods, such as `notify_`. To avoid code bloat, our transformation adds these methods to the topmost class of each inheritance hierarchy in an application.)

Let's now consider the second problem: maintaining thread identity over the network. Figure 7 demonstrates how using the thread id equivalence classes can solve the "zigzag deadlock problem" presented above.

These thread id equivalence classes enable 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.)

```
private synchronized void enter () {
    while (_timesLocked != 0 && curThreadEqClass != _holderThreadId)
```

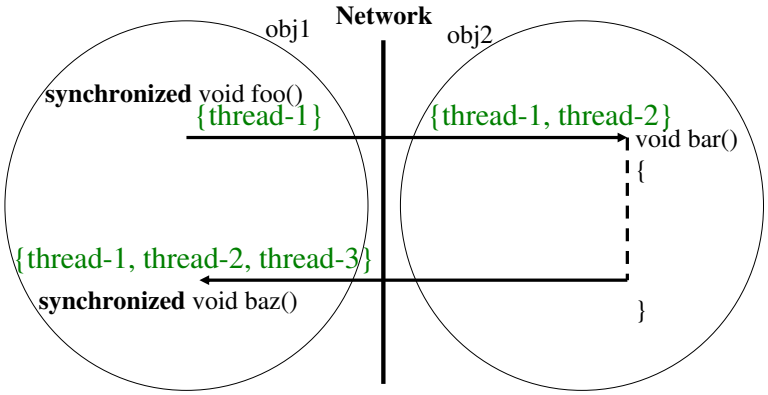

Fig. 7. Using thread id equivalence classes to solve the "zigzag deadlock problem" in Java RMI.

```
        try {
            wait();
        } catch(InterruptedException e) {...}

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

This causes threads that are not in the equivalence class of the holder thread to wait until the monitor is unlocked.

5.3 Efficiency and Past Approaches

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. In this way, the rest of the application does not need to be modified at all to propagate thread equivalence classes: only the code performing the distributed communication changes.

Adding custom logic to RMI stubs can be done by modifying the RMI compiler, but this would negate J-Orchestra's 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.)

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

    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.

The J-Orchestra solution is not the first in the design space of enabling monitor-style synchronization over middleware. Past solutions fall in two different camps. A representative of the first camp is the approach of Haumacher et al. [2003], which proposes a replacement of Java RMI that maintains thread identity over the network. Employing special-purpose middleware is undesirable, however, for reasons of portability and ease of deployment. Indeed, using standard middleware is one of the primary design objectives of J-Orchestra. The second camp, represented by the work of Weyns et al. [Weyns et al. 2002; 2004], advocates transforming the client application, as in the J-Orchestra approach. Since clients do not know whether a method they call is local or remote, the Weyns et al. solution consists of conservatively re-writing all method calls in the application to pass one extra parameter—the thread identifier. This is quite inefficient compared to the J-Orchestra approach of only modifying RMI stubs.[6] As detailed elsewhere [Tilevich and Smaragdakis

[6]A second disadvantage of the Weyns et al. approach compared to our solution is that adding

2004], we found the overhead of the Weyns et al. technique to be between 5.5 and 12% of the total execution time for SPEC JVM'98 benchmark applications. In contrast, the J-Orchestra technique has a zero overhead for actual benchmark applications, and an overhead below 4% even in the worst-case micro-benchmark of a do-nothing method over an infinitely fast network (local call).

5.4 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. 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.

6. APPLICABILITY AND DISCUSSION

The previous sections presented in detail the most interesting technical aspects of J-Orchestra and its transformation machinery. Armed with this understanding, we can next try to raise the level of discussion and capture the broader picture of J-Orchestra's applicability and value. This continues the user-level argument and examples of Section 2, but with a more precise and general treatment.

---

an extra argument is not valid for methods that may be called by unmodifiable native code, or through standard interfaces. E.g., 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.

Applicability

Automatic partitioning is not a substitute for general distributed systems development. The important insight about the approach is not that it is widely applicable, but that it can be applicable to a non-trivial subset of applications. Correctness is not a primary issue: With a small number of exceptions, discussed in Section 4, J-Orchestra can handle a large subset of the Java language, and, as a consequence, can correctly partition a large class of realistic unsuspecting applications. Among these, however, partitioning with J-Orchestra will prove useful only for well-defined cases.

The main issue for automatically partitioned applications is performance. There are several aspects of the J-Orchestra performance that can be measured. In the past [Tilevich and Smaragdakis 2002b; 2004; Tilevich et al. 2005], we reported results on the overhead of adding proxies, of registering remote objects, of maintaining concurrency semantics, etc. We also compared to other distributed execution or monitoring techniques to demonstrate end-to-end benefit. The J-Orchestra argument in its core is not about such low-level performance differences, however. In our successfully partitioned applications, the overheads of J-Orchestra indirection are unmeasurably low: Co-anchored objects access each other directly, and only a tiny fraction of the program's references incur any overhead. The real overheads are not part of the J-Orchestra technology, but concern the unavoidable cost of network communication. Procedure calls over a middleware mechanism like Java RMI are many thousands of times slower than local calls. Most of this overhead cannot be eliminated, as it hinges on network latency and bandwidth limitations. Thus, the fundamental question that determines whether an application can be efficiently partitioned with J-Orchestra has to do with the application's locality, under its original structure. Namely, can the application tolerate making communication between some of its parts thousands of times slower?

To describe the applications that can be successfully partitioned, we draw inspiration from parallel processing, in which the term *embarrassingly parallel* describes problems that can be seamlessly segmented into parallel tasks, with little inter-task communication. By analogy, we introduce the term *embarrassingly loosely coupled* to describe applications that can benefit from automatic partitioning with J-Orchestra. Such applications must satisfy two criteria:

(1) they must consist of components that exchange little data with the rest of the application

(2) these components must be statically identifiable by examining the structure of the application code at the class or the module level.

By examining the relationship among application classes, the user of J-Orchestra (aided by our analysis tools) should be able to identify distinct components comprising multiple classes. Then, during run-time, the data coupling among these components should be very small. In other words, an application should have very clear communication and locality patterns. Since the application logic will not change after partitioning has taken place, a large number of remote accesses will be detrimental to performance.

Most partitioned applications will take longer to run, and for some applications

naïve partitioning can render them unusable. To improve the performance of a partitioned application, various optimizations can reduce the number of remote calls such as using an optimal static class placement and using object mobility. In modern networks, in which improvements in bandwidth have surpassed those in latency, object mobility offers great potential benefit. (I.e., it is faster to move much of the data that a remote method will operate on and move it back on method return, rather than have the method send remote messages to read or update data values.)

To obtain a reasonable estimate of whether partitioning an application is likely to incur a prohibitively large performance overhead, online and offline profile techniques should take into consideration both the initial static placement of objects and their potential mobility. Having multiple profiling samples can help the programmer decide whether the incurred overhead is acceptable.

Preconditions

In addition, unmodifiable code (e.g., OS or JVM code) should not use objects shared among partitions. Otherwise, the application structure must change to make partitioning possible. Also, the resulting distributed application should primarily have synchronous communication patterns. If good performance or reliability requires asynchronous communication, the application structure must change.

Embarrassingly loosely coupled applications can be partitioned automatically without significant loss in performance due to network communication. However, in order to get any benefit, the application needs to have a reason to be distributed. The foremost reason for distributing an application with J-Orchestra is to take advantage of remote hardware or software resources (e.g., a processor, a database, a graphical screen, or a sound card).

Of course, one reason to partition an application is to take advantage of parallelism. Distinct machines will have distinct CPUs. If the original centralized application is multi-threaded, we can use multiple CPUs to run threads in parallel. Although distribution-for-parallelism is a potential application of J-Orchestra, we have not examined this space so far. The reason is that parallel applications either are written to run in distributed memory environments in the first place, or have tightly coupled concurrent computations.

These and other limitations make automatic partitioning particularly well suited for *resource-driven distribution*, especially when the application has distinct parts that each deal with different hardware or software resources. A typical resource-driven distribution scenario is a centralized application needing to access resources located on one or more remote machines scattered around the network, each possessing unique resources, such as a large graphical display screen, high-quality speakers, a digital camera, etc. In most cases, the different parts of such an application are loosely coupled. Although network communication can be a bottleneck, most successful applications of automatic partitioning can achieve high performance for loosely coupled applications by placing the code near the resource it accesses. The gains of placing the relevant code near a resource are precisely the reason for partitioning an application, rather than executing it remotely in its entirety.

To summarize, we can characterize the domain of J-Orchestra as *partitioning embarrassingly loosely coupled applications for resource-driven distribution.*

7. RELATED WORK

Much research work is closely related to J-Orchestra, either in terms of goals or in terms of methodologies, and we discuss some of this work next.

Several recent systems other than J-Orchestra can also be classified as automatic partitioning tools. In the Java world, the closest approaches are the Addistant [Tatsubori et al. 2001] and Pangaea [Spiegel 2000; 2002] systems. The Coign system [Hunt and Scott 1999] has promoted the idea of automatic partitioning for applications based on COM components.

All three systems do not address the problem of distribution in the presence of unmodifiable code. Coign is the only one of these systems to have a claim at scalability, but the applications partitioned by Coign consist of independent components to begin with. Coign does not address the hard problems of application partitioning, which have to do with pointers and aliasing: components cannot share data through memory pointers. Such components are deemed non-distributable and are located on the same machine. Practical experience with Coign showed that this is a severe limitation for the only real-world application included in Coign's example set (the Microsoft PhotoDraw program). The overall Coign approach would not be feasible for applications written in a general-purpose language (like Java, C, C#, or C++) in which pointers are prevalent, unless these applications have been developed following a strict component-based implementation methodology.

The GARF system [Guerraoui et al. 1997] uses automatic program distribution to simplify the creation of reliable distributed applications. GARF was implemented in Smalltalk on top of the Isis group communication system [Birman and Van Renesse 1994]. The GARF runtime uses the reflective capabilities of Smalltalk [Goldberg and Robson 1983] to introduce distribution and replication to centralized applications. Compared to the static binary rewrite of J-Orchestra, the dynamic approach of the GARF system's distribution is on the other end of the design space. Nevertheless, GARF's approach of achieving reliability by replicating critical components over multiple machines is a promising future work direction in automatic partitioning.

The Pangaea system [Spiegel 2000; 2002] has very similar goals to J-Orchestra. Pangaea, however, includes no support for making Java system classes remotely accessible. Thus, Pangaea cannot be used for resource-driven distribution, as most real-world resources (e.g., sound, graphics, file system) are hidden behind system code. Pangaea utilizes interesting static analyses to aid partitioning tasks (e.g., object placement) but these analyses ignore unmodifable (system) code.

The JavaParty [Haumacher et al. 2003; Philippsen and Zenger 1997] system is closely related to J-Orchestra. The similarity is not so evident in the objectives, since JavaParty only aims to support manual partitioning and does not deal with system classes. However, the implementation techniques of JavaParty are very similar to the ones of J-Orchestra, especially for the newest versions of JavaParty. For distributed synchronization, JavaParty relies on KaRMI, a drop-in replacement for RMI, that maintains correct multithreaded execution over the network efficiently. In contrast, J-Orchestra implements distributed synchronization on top of standard middleware.

J-Orchestra bears similarity with such diverse systems as DIAMONDS [Craig

et al. 1993], FarGo [Holder et al. 1999], and AdJava [Fuad and Oudshoorn 2002]. DIAMONDS clusters are similar to J-Orchestra anchored and mobile groups. FarGo groups are similar to J-Orchestra anchored groups. Notably, however, FarGo has focused on grouping classes together and moving them as a group. In fact, groups of J-Orchestra objects that are all anchored by choice could well move, as long as all objects in the group move. We have not yet investigated such mobile groups, however.

The pioneering work at MCC in the early 90's identified classes as suitable entities for performing resource allocation in distributed systems. The experimental system described by Chatterjee [1992] uses class profiling as a guide for assigning objects to the nodes of a distributed system. J-Orchestra has fully explored the idea of resource-based partitioning at the class or group of classes level of granularity, demonstrating the feasibility and scalability of the approach.

Automatic partitioning is essentially a distributed shared memory (DSM) technique. Nevertheless, automatic partitioning differs from traditional DSMs in several ways. First, automatic partitioning systems such as J-Orchestra do not change the runtime system, but only the application. Traditional DSM systems like Munin [Carter et al. 1991], Orca [Bal et al. 1998; Bal and Kaashoek 1993], and, in the Java world, cJVM [Aridor et al. 1999; Aridor et al. 2000], and Java/DSM [Yu and Cox 1997] use a specialized run-time environment in order to detect access to remote data and ensure data consistency. Also, DSMs have usually focused on parallel applications and require programmer intervention to achieve high-performance. In contrast, automatic partitioning concentrates on resource-driven distribution, which introduces a new set of problems (e.g., the problem of distributing around unmodifable system code, as discussed earlier). Among distributed shared memory systems, the ones most closely resembling the J-Orchestra approach are object-based DSMs, like Orca [Bal et al. 1998; Bal and Kaashoek 1993].

Mobile object systems, like Emerald [Black et al. 2007] have formed the inspiration for many of the J-Orchestra ideas on object mobility scenarios. The novelty of J-Orchestra is not in the object mobility ideas but in the rewrite that allows them to be applied to an oblivious centralized application.

A promising direction for future work in automatic partitioning would be to give the user more control and enable more drastic modifications of the original application. For instance, the proxying approach of J-Orchestra should be a good fit for adding flexibility in configuring the communication protocol, as in the BAST system [Garbinato and Guerraoui 1997]. This flexibility would allow choosing an unreliable (but more efficient) communication protocol on a per-case basis. At the same time, such flexibility would fit well with another promising future direction: that of allowing the replication of components from the original centralized application and transforming a single call into a multicast message in the distributed application. Such transformations open the door for automatic partitioning to handle a much larger space of distributed applications, at the expense of requiring more user intervention.

Approaches to richer middleware can simplify the implementation of the J-Orchestra distributed synchronization approach. For instance, DADO [Wohlstadter et al. 2003; Wohlstadter and Devanbu 2006] enables passing custom information

between client and server of a remote call. This would be an easy alternative to our custom bytecode transformations of stubs and skeletons. Nevertheless, using DADO would not eliminate the need for bytecode transformations that replace monitor control methods and synchronization blocks.

Both the D [Lopes 1997] and the Doorastha [Dahm 2000] systems allow the user to easily annotate a centralized program to turn it into a distributed application. Although these systems are higher-level than explicit distributed programming, they are significantly lower-level than J-Orchestra. The entire burden is shifted to the programmer to specify which semantics is valid for a specific class (e.g., whether objects are mobile, whether they can be passed by-copy, and so forth). Programming in this way requires complete understanding of the application behavior and can be error-prone: a slight error in an annotation may cause insidious inconsistency errors.

8. CONCLUSIONS

This article has discussed J-Orchestra, a software system and broader project on separating distribution concerns. The goal of this project has been to investigate whether software tools working with standard mainstream languages, systems software, and virtual machines can effectively and efficiently separate distribution concerns from application logic for object-oriented programs. We believe that this work will contribute to the development of versatile tools and technology with practical value, innovative designs, and the potential to become mainstream in the future.

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

REFERENCES

Aridor, Y., Factor, M., and Teperman, A. 1999. cJVM: A single system image of a JVM on a cluster. In *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*. IEEE Computer Society, Washington, DC, USA, 4.

Aridor, Y., Factor, M., Teperman, A., Eilam, T., and Schuster, A. 2000. A high performance cluster JVM presenting a pure single system image. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*. ACM Press, New York, NY, USA, 168–177.

Atkinson, M. P., Daynés, L., Jordan, M. J., Printezis, T., and Spence, S. 1996. An orthogonally persistent Java. *SIGMOD Rec. 25,* 4, 68–75.

Bal, H. E., Bhoedjang, R., Hofman, R., Jacobs, C., Langendoen, K., Rühl, T., and

Kaashoek, M. F. 1998. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems 16,* 1, 1–40.

Bal, H. E. and Kaashoek, M. F. 1993. Object distribution in Orca using compile-time and run-time techniques. *SIGPLAN Not. 28,* 10, 162–177.

Birman, K. P. and Van Renesse, R. 1994. *Reliable distributed computing with the Isis toolkit.* IEEE Computer Society Press, Los Alamitos, Calif.

Birrell, A. D. and Nelson, B. J. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems 2,* 1, 39–59.

Black, A. P., Hutchinson, N. C., Jul, E., and Levy, H. M. 2007. The development of the Emerald programming language. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of Programming Languages.* 11–1–11–51.

Carter, J. B., Bennett, J. K., and Zwaenepoel, W. 1991. Implementation and performance of Munin. In *SOSP '91: Proceedings of the thirteenth ACM Symposium on Operating Systems Principles.* ACM Press, New York, NY, USA, 152–164.

Chatterjee, A. 1992. The Class is an Abstract Behaviour Type for Resource Allocation of Distributed Object-Oriented Programs. In *unpublished paper presented at the OLDA-2 workshop at OOPSLA-92.*

Craig, G., Bellur, U., Shank, K., and Lea, D. 1993. Clusters: A Pragmatic Approach Towards Supporting a Fine Grained Active Object Model in Distributed Systems. In *The 9th International Conference on Systems Engineering.*

Dahm, M. 1999. Byte code engineering. In *Proceedings of JIT.*

Dahm, M. 2000. Doorastha: a step towards distribution transparency. In *Proceedings of JIT.*

Dijkstra, E. 1982. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, 60–66.

Fuad, M. M. and Oudshoorn, M. J. 2002. AdJava: automatic distribution of Java applications. In *ACSC '02: Proceedings of the twenty-fifth Australasian conference on Computer science.* Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 65–75.

Garbinato, B. and Guerraoui, R. 1997. Using the strategy design pattern to compose reliable distributed protocols. In *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS).* USENIX Association, Berkeley, CA, USA, 17–17.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80 : the language and its implementation.* Addison-Wesley series in computer science. Addison-Wesley, Reading, Mass.

Guccione, S., Levi, D., and Sundararajan, P. 1999. JBits: A Java-based Interface for Reconfigurable Computing. In *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD).*

Guerraoui, R., Garbinato, B., and Mazouni, K. 1997. Garf: A tool for programming reliable distributed applications. *IEEE Parallel Distrib. Technol. 5,* 4, 32–39.

Haumacher, B., Moschny, T., Reuter, J., and Tichy, W. F. 2003. Transparent distributed threads for Java. In *Parallel and Distributed Processing Symposium, IPDPS 2003.*

Holder, O., Ben-Shaul, I., and Gazit, H. 1999. Dynamic layout of distributed applications in FarGo. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA, USA, 163–173.

Hunt, G. C. and Scott, M. L. 1999. The Coign automatic distributed partitioning system. In *OSDI '99: Proceedings of the third symposium on Operating Systems Design and Implementation.* USENIX Association, Berkeley, CA, USA, 187–200.

Jones, R. and Lins, R. 1996. *Garbage collection : algorithms for automatic dynamic memory management.* Wiley, Chichester, [Eng.]; New York.

Kienzle, J. and Guerraoui, R. 2002. AOP: Does it Make Sense? The Case of Concurrency and Failures. In *European Conference on Object-Oriented Programming (ECOOP).* Springer, 37–61.

Lea, D. 1997. *Concurrent programming in Java: design principles and patterns.* Addison Wesley.

Liogkas, N., MacIntyre, B., Mynatt, E. D., Smaragdakis, Y., Tilevich, E., and Voida, S. 2004. Automatic partitioning: Prototyping ubiquitous-computing applications. *IEEE Pervasive Computing 03,* 3, 40–47.

Lopes, C. 1997. D: A Language Framework For Distributed Programming. Ph.D. thesis, Northeastern University.

Philippsen, M. and Zenger, M. 1997. JavaParty—transparent remote objects in Java. *Concurrency: Practice and Experience 9,* 11, 1225–1242.

Richardson, T., Stafford-Fraser, Q., Wood, K. R., and Hopper, A. 1998. Virtual network computing. *IEEE Internet Computing 2,* 1, 33–38. 1089-7801.

Scheifler, R. 1987. RFC 1013 X Window System Protocol, Version 11.

Scheifler, R. W. and Gettys, J. 1986. The X window system. *ACM Transactions on Graphics 5,* 2, 79–109.

Shapiro, M. 1986. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the* 6th *International Conference on Distributed Computing Systems*. 198–204.

Spiegel, A. 2000. Automatic Distribution in Pangaea. In *CBS 2000*.

Spiegel, A. 2002. Automatic Distribution of Object Oriented Programs. Ph.D. thesis, FU Berlin, FB Mathematik und Informatik.

Tatsubori, M., Sasaki, T., Chiba, S., and Itano, K. 2001. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *European Conference on Object-Oriented Programming (ECOOP)*.

Tilevich, E. 2005. Software Tools for Separating Distribution Concerns. Ph.D. thesis, Georgia Institute of Technology, College of Computing.

Tilevich, E. and Smaragdakis, Y. 2002a. Automatic application partitioning: The J-Orchestra approach. In *ECOOP 2002 Workshop on Mobile Object Systems*.

Tilevich, E. and Smaragdakis, Y. 2002b. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, LNCS 2374, 178–204.

Tilevich, E. and Smaragdakis, Y. 2004. Portable and efficient distributed threads for Java. In *ACM/IFIP/USENIX Middleware Conference*. Springer-Verlag, 478–492.

Tilevich, E. and Smaragdakis, Y. 2006. Transparent Program Transformations in the Presence of Opaque Code. In *Generative Programming and Component Engineering (GPCE) Conference*. 89–94.

Tilevich, E., Smaragdakis, Y., and Handte, M. 2005. Appletizing: Running legacy Java code remotely from a Web browser. In *International Conference on Software Maintenance (ICSM)*.

Waldo, J., Wollrath, A., Wyant, G., and Kendall, S. 1994. A Note on Distributed Computing. Tech. rep., Sun Microsystems, Inc. Mountain View, CA, USA.

Weyns, D., Truyen, E., and Verbaeten, P. 2002. Distributed Threads in Java. In *The International Symposium on Distributed and Parallel Computing (ISDPC)*.

Weyns, D., Truyen, E., and Verbaeten, P. 2004. Serialization Of Distributed Threads In Java. *Scalable Computing: Practice and Experience 6,* 1, 81–98.

Wohlstadter, E. and Devanbu, P. 2006. Aspect-oriented development of crosscutting features in distributed, heterogeneous systems. In *Transactions on Aspect-Oriented Software Development II*. Springer-Verlag, 69–100.

Wohlstadter, E., Jackson, S., and Devanbu, P. 2003. DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 174–186.

Wollrath, A., Riggs, R., and Waldo, J. 1996. A distributed object model for the Java^{TM} system. In *COOTS'96: Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*. USENIX Association, Berkeley, CA, USA, 17–17.

Yu, W. and Cox, A. 1997. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience 9,* 11, 1213–1224.

...