Appletizing: Running Legacy Java Code Remotely From a Web Browser

Eli Tilevich, Yannis Smaragdakis
College of Computing, Georgia Institute of Technology
{tilevich, yannis}@cc.gatech.edu

Marcus Handte
University of Stuttgart[1]
m.handte@web.de

Abstract

Adding distributed capabilities to existing programs has come to the forefront of software evolution. As a standard Java distributed technology, applets offer the advantages of being easily deployable over web browsers and requiring little to no explicit distributed programming. Yet applets are inflexible: they download remote code and run it only on the client machine. We present appletizing: a semi-automatic approach to transforming a Java GUI application into a client-server application, in which the client runs as a Java applet that communicates with the server through RMI. To enable appletizing, we have expanded the capabilities of J-Orchestra, our automatic partitioning system that takes as input a Java application in bytecode format and transforms it into a distributed application, running across multiple standard JVMs. We discuss the motivation, benefits, and J-Orchestra support for appletizing, and validate our approach via a set of case studies and associated benchmarks.

1. Introduction

As the emergence of the Internet has changed the computing landscape, distribution is no longer optional but necessary in a large and growing number of software systems. The focus of distributed computing has been shifting from “distribution for parallelism” to “resource-driven distribution,” with the resources of an application naturally remote from each other or from the computation. Because of this shift, more and more legacy code needs to be adapted for distributed execution. In out context, the term ‘legacy’ refers to all centralized Java applications, written without distribution in mind, that need to be changed to move parts of their execution to a remote machine. The amount of such legacy code in Java is by no means insignificant with the Java technology being a decade old and four million Java developers worldwide [4].

A large part of what makes Java a language that “allows application developers to write a program once and then be able to run it everywhere on the Internet” [9] are standard distribution technologies over the web. Such Java technologies as applets and servlets have two major advantages: they require little to no explicit distributed programming and they are easily deployable over standard web browsers. Nevertheless, these technologies are inflexible. In the case of applets, a web browser first transfers an applet’s code from the server site to the user system and then executes it safely on its Java Virtual Machine (JVM), usually in order to draw graphics on the client’s screen. In the symmetric case of servlets, code executes entirely on the server, usually in order to access a shared resource such as a database, transmitting only simple inputs and outputs over the network. Therefore, these standard technologies offer a hard-coded answer to the important question of how the distribution should take place, and it is the same for each applet and servlet. Besides these two extremes, one can imagine many other solutions that are customizable for individual programs. A hybrid of the two approaches promises significant flexibility benefits: the programmer can leverage both the resources of the client machine (e.g., graphics, sound, mouse input) and the resources of a server (e.g., shared database, file system, computing power). At the same time, the application will be both safe and efficient: one can benefit from the security guarantees provided by Java applets, while communicating only a small amount of data between the applet and a remote server.

The challenge is to get an approach that runs code both on clients and on a server while avoiding explicit distributed systems development, just like applet and servlet technologies do. This paper presents *appletizing*: a semi-automatic approach to transforming a centralized, monolithic Java GUI application into a client-server application, in which the client runs as a Java applet that communicates with the server through Java RMI. Appletizing builds upon automatic partitioning, a technology in which a tool takes as input a regular program and user-supplied location information for its code and data, and automatically rewrites the program so that both the code and the data divide into parts that can run in the desired location. Any data exchange between parts of the program at differ-

1. This research was performed while the author was at the Georgia Institute of Technology.

ent locations automatically becomes remote communication.

Because appletizing is essentially a specialization of automatic partitioning with a predefined deployment environment for the resulting client-server applications, we implemented it on top of J-Orchestra [21][22], our automatic partitioning system, that takes as input a Java application in bytecode format and transforms it into a distributed application, running across multiple JVMs. Similarly to regular partitioning, appletizing requires no explicit programming or modification to the JVM or its standard runtime classes.

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

Overall, our approach offers a unique combination of the following benefits:

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

2. J-Orchestra Overview

As a special-purpose application of our J-Orchestra automatic partitioning system, appletizing is made possible by the hallmark ability of J-Orchestra to deal correctly with system code: the code split during appletizing is system code that deals with system resources such as graphics and file storage. Therefore, we begin by presenting J-Orchestra and its program transformations for distribution.

2.1. Technical Overview

J-Orchestra is a GUI-enabled tool that, under human guidance, handles all the tedious tasks of splitting the functionality of a centralized application into distinct entities running across different network sites. First, the system lists all application classes and the systems classes they reference. Then, the user creates different "sites" and (at a first approximation) assigns classes to sites. In the end, J-Orchestra rewrites the application to produce distinct partitions that can be run on separate machines, on standard, unmodified Java VMs. J-Orchestra relieves its users of the necessity to change the application source code (or even have source code available), to deal with middleware directly, and to understand all the potentially complex data sharing structure of the application. For a large subset of Java, the partitioned application is guaranteed to behave exactly like its original, centralized version.

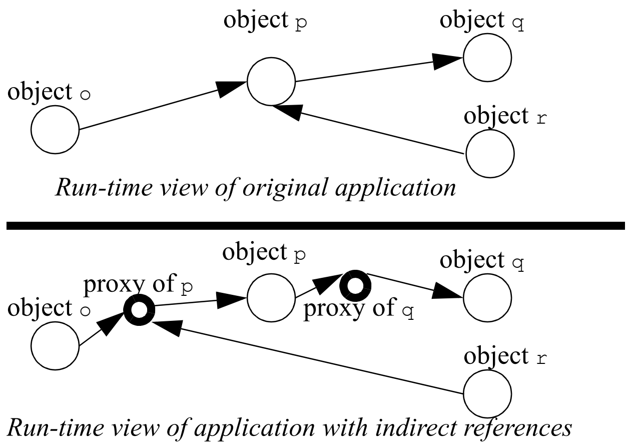
object p
object q
object o
object r
Run-time view of original application
object p
object q
proxy of p
object o
proxy of q
object r
Run-time view of application with indirect references

Figure 1: Indirect referencing schematically. Proxy objects could point to their targets either locally or over the network.

To maintain correct execution under distributed memory spaces, the J-Orchestra rewrite follows the standard technique of adding proxies to convert all direct object references to indirect ones. Proxies hide the location of objects creating an abstraction of shared memory, which is necessary for correct execution of the program across different machines in the presence 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 1 shows schematically the effects of the indirect referencing approach, which has been used in several prior systems [19][20][23].

Adding indirection without changing the JVM entails rewriting the code of the partitioned application. Thus, when the original application would create a new object, the partitioned application will also create a proxy and return it; whenever an object in the original application would access another object's fields, the corresponding object in the partitioned application would have to call a method in the proxy to get/set the field data; whenever a method would be called on an object, the same method now needs to be called on the object's proxy; etc.

The difficulty of this rewrite approach is that it needs to be applied to *all code that might hold references to remote objects*, which is not only the application code, but also the code inside the runtime system. In the case of the Java VM, such code is encapsulated by system classes that control various system resources through native code in the JVM binary (executable or dynamic libraries). JVM 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 JVM code that performs a file operation to make it access the file object correctly for both local and remote files. 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 likely occur (e.g., because the native code will try to read fields directly from the object). (For simplicity, we assume the application itself does not contain native code—i.e., it is a "pure Java" application.)

The conceptual novelty of J-Orchestra (compared to past partitioning systems [13][20][23] and distributed shared memory systems [1][2][5][24]) consists of addressing the problems resulting from inability to analyze and modify Java VM code. Prior partitioning systems have ignored the issues arising from native system code. J-Orchestra features a novel rewrite mechanism that ensures that, at run-time, references are always in the expected form ("direct" = local or "indirect" = possibly remote) for the code that handles them. The result is that J-Orchestra can split code that deals with system resources, safely running, e.g., all sound synthesis code on one machine, while leaving all graphics code on another.

Due to lack of space and previous publication [21], our discussion of J-Orchestra in this paper is slightly simplified and omits some interesting elements. These include:

- a type-based "classification" heuristic that groups classes whose instances can be accessed by the same native code. Although by nature this analysis cannot be sound (native code can potentially access all application objects) in practice it is valuable in helping the user decide groupings for classes that should be co-located. The groupings typically reflect distinct resources, e.g., classes that deal with graphics, classes that deal with sound, and classes that deal with files end up in three distinct groups.
- optimizations for creating remote objects lazily, i.e., when the object first gets accessed remotely.
- the handling of Java language features, such as static methods, inner classes, inheritance, etc.
- limitations of the system: unsupported language features include reflective field access, dynamic loading, volatile variables, and more. Prior limitations [21] with respect to multithreading and monitor-style synchronization have been addressed and the J-Orchestra distributed threading mechanism is described in a recent publication [22].

2.2. The J-Orchestra Rewrite

Appletizing builds upon the J-Orchestra rewrite that enables remote access to JVM resources, such as graphics, file I/O, and sound. To accomplish such remote access, J-Orchestra distinguishes between two different kinds of classes: *anchored* and *mobile*. While "anchored" objects remain in a single JVM for their entire lifetime, mobile objects can migrate from site to site at run-time.

The two reasons behind "anchoring" classes are preserving correctness and improving performance. A class must be anchored if its objects could be accessed through native code, which also determines where such objects should be anchored (i.e., if an object can be accessed by native code running on some machine, the object should be anchored there). In addition, anchoring a class *by choice* can eliminate the overhead of accessing its objects in local code on a specific site (i.e., make the access as quick as in the original centralized application). In a typical J-Orchestra partitioning, the vast majority of objects are anchored by choice. Anchored objects can still be accessed indirectly (through a proxy) from other machines and by mobile objects even when these happen to be on the same machine.

The J-Orchestra "rewrite engine" is responsible for transforming existing application code through bytecode manipulation (we use BCEL [6] for bytecode engineering) and generating new code to turn a centralized application into a distributed one. We outline several major steps of the J-Orchestra rewrite process next.

Some transformations are at the bytecode level. One example is ensuring that all data exchange among potentially remote objects is done through method calls: every time an object reference accesses fields of a different object and that object is either mobile or anchored on a different site, the corresponding instructions are replaced with a method invocation that will get/set the required data. Another example is transforming original application classes into remote ones that extend the Java RMI class `UnicastRemoteObject` and can be registered as RMI remote objects (i.e., can be passed by-reference over the network).

In addition to bytecode rewriting, J-Orchestra also generates some code from scratch, such as a proxy and a remote interface (i.e., extending `java.rmi.Remote`) for each class in the application. These generated classes define all the methods as in the original class. A J-Orchestra proxy is essentially a delegate for a remote class or its RMI "stub," providing a mechanism for remote execution. We show below a simplified version of the code generated for a class `A`.

```
//Original mobile class A
class A {
 void foo () { ... }
}
//Proxy for A (generated in source code form)
class A implements java.io.Externalizable {
 //ref at different points can point to either
 //remote implementation directly or RMI stub.
 A__interface ref;
 ...
 void foo () {
  try { ref.foo (); } catch (RemoteException e) {
    //let user provide custom failure handling
  }
 }//foo
}//A
//Interface for A (generated in source code form)
interface A__interface extends java.rmi.Remote {
 void foo () throws RemoteException;
}
//Remote implementation (produced in bytecode
//form by modifying original class A)
class A__remote extends UnicastRemoteObject
              implements A__interface {
 void foo () throws RemoteException {...}
}
```

In addition, proxies provide logic for various other pieces of functionality. First, they contain globally unique identifiers, through which the J-Orchestra runtime system maintains an "at most one proxy per site" invariant. Also, proxies manage their own serialization (i.e., implement `java.io.Externalizable`), providing a mechanism for object mobility that can move objects during serialization as specified by a custom mobility scenario. Finally, proxies are generated as source code to enable the sophisticated user to supply custom handling code for remote errors.

Because anchored classes are accessed directly by their co-anchored clients (i.e., classes anchored on the same site), they cannot change their superclass (to `UnicastRemoteObject`) and must use a different mechanism to enable remote execution. An extra level of indirection is added through special purpose classes called *translators*, which implement remote interfaces and make anchored classes look like mobile classes as far as the rest of the J-Orchestra rewrite is concerned. Regular proxies, as well as remote versions are created for translators, exactly like for mobile classes.

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 and vice versa, as these references get passed between anchored and mobile code. Consider what happens when references to anchored objects are passed from mobile code to anchored code. For instance, in Figure 2, a mobile application object `o` holds a reference `p` to an object of type `java.awt.Point`. Object `o`

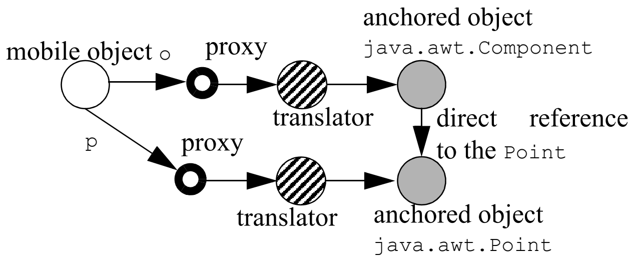

Figure 2: Mobile code refers to anchored objects indirectly (through proxies) but anchored code refers to the same objects directly. Each kind of reference should be derivable from the other.

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 and, thus, expects a direct reference to a `java.awt.Point` (for instance, so it can assign it or compare it with a different reference). In general, the two kinds of references should be implicitly convertible to each other at run-time, depending on what kind is expected by the code currently being run.

Translation takes place when a method is called on an anchored object. The translator implementation of the method "unwraps" all method parameters (i.e., converts them from indirect to direct) and "wraps" all results (i.e., converts them from direct to indirect). Since all data exchange between mobile code and anchored code happens through method calls (which go through a translator) we can be certain that references are always of the correct kind.

Past systems that follow a similar rewrite as J-Orchestra [11][19][20][23] do not offer a translation mechanism. The partitioned application is safe only if objects passed to system code are guaranteed to always be on the same site as that code. This is a big burden to put on the user. The translation mechanism of J-Orchestra ensures that all the interactions between application and system code are in the right form, making appletizing possible.

3. Supporting Appletizing

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). Several special-purpose technologies do this already: distributed file systems allow storage on remote disks; remote desktop applications (e.g., VNC, X) allow transferring graphical data from a remote machine; network printer protocols let users print remotely. Nonetheless, the advantage of automatic partitioning is that it can put the code near the resource that it controls. Specifically, partitioning makes it possible to draw graphics locally on the client machine from less data than it takes to transfer the entire graphical representation over the network, while collocating the server resources with the code that controls them. As a special kind of partitioning, appletizing not only offers the same benefits but also provides a higher degree of automation. The J-Orchestra mechanisms that make this automation possible are static analysis and profiling that in addition to bytecode rewriting and runtime services enable appletizing. We describe them in turn next.

3.1. Static Analysis for Appletizing

Consider an arbitrary centralized Java AWT/Swing application that we want to transform into a client-server application through appletizing. First, we classify the application's code (both application classes and the referenced JRE system classes) into four distinct groups, as Figure 3 demonstrates schematically.

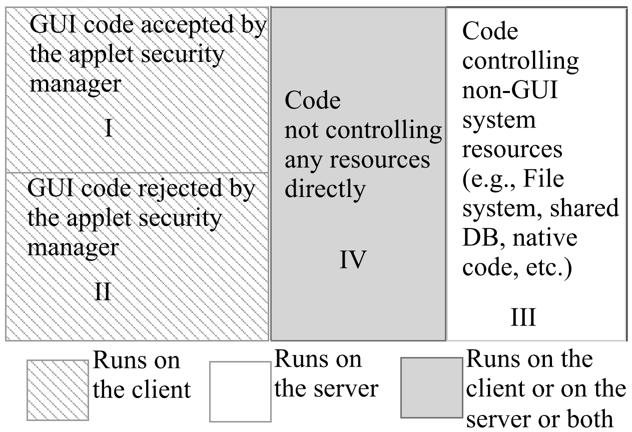
GUI code accepted by the applet security manager
I
GUI code rejected by the applet security manager
II
Code not controlling any resources directly
IV
Code controlling non-GUI system resources (e.g., File system, shared DB, native code, etc.)
III
Runs on the client
Runs on the server
Runs on the client or on the server or both

Figure 3: The appletizing perspective code view of a centralized Java GUI application.

Group I contains the GUI classes that can safely execute within an applet. Group II contains the GUI classes whose code include instructions that the applet security manager prevents from executing within an applet. For example, an applet cannot perform disk I/O. Group III contains the classes that must execute on the server. The classes in this group control various non-GUI system resources that applets are not allowed to access, such as file I/O operations, shared resources (e.g., a database), and native (JNI) code. Group IV contains the classes that do not control any system resources directly and as such can be placed on either the client or the server sites, purely for performance reasons. Moreover, objects of classes in this group do not have to remain on the same site during the execution of the program: they can migrate on demand, or according to an application-specific pattern.

We implemented the analysis of classes for appletizing on top of the standard J-Orchestra type-based "classification" heuristic that groups classes whose instances can be accessed by the same native code. At a first approximation, the heuristic examines the application bytecode files to see which class types get passed as arguments to system code, and groups these classes together with their subclasses and the native code front-end classes in an anchored group. Access through interfaces is safe even when the class is replaced by a proxy, so it does not entail any constraints in the analysis. Since the heuristic is type-based it would not be safe if type information were obscured (e.g., if a method accepted an Object type and used reflection to determine if the object is suitable). However, we did not find this to be an issue in practice.

3.2. Profiling for Appletizing

Having completed the aforementioned classification heuristics, J-Orchestra assigns the classes in groups I, II, and III to the client, client, and server sites, respectively. The classification does not offer any help in assigning the classes in group IV, so the user has to do this manually before the rewriting for appletizing can commence. Deciding on the location of a class just by looking at its name can be a prohibitively difficult task, particularly if no source code is available and the user has only a black-box view of the application. To help the user in determining a good placement, J-Orchestra offers an off-line profiler that reports data exchange statistics among different entities (i.e., anchored groups and mobile classes). Integrated with the profiler is a clustering heuristic that, given some initial locations and the profiling results, determines a good placement for all classes. The heuristic is strictly advisory—the user can override it at will. Our heuristic implements a greedy strategy: start with the given initial placement of a few entities and compute the affinity of each unassigned entity to each of the locations. (Affinity to a location is the amount of data exchanged between the entity and all the entities already assigned to the location combined.) Pick the overall maximum of such affinity, assign the entity that has it to the corresponding location and repeat until all entities are assigned. In principle, appletizing offers more opportunities than general application partitioning for automation in clustering: optimal clustering for a client-server partitioning can be done in polynomial time, while for 3 or more partitions the problem is NP-hard. In practice we have not yet had the need to replace our heuristic for better placement.

In terms of implementation, the J-Orchestra profiler has evolved through several incarnations. The first profiler worked by instrumenting the Java VM through the JVMPI and JVMDI (Java Virtual Machine Profiling/Debugging

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

An important issue with profiling concerns the use of off-line vs. on-line profiling. Several systems with goals similar to ours (e.g., Coign [13] and AIDE [18]) use on-line profiling in order to dynamically discover properties of the application and possibly alter partitioning decisions on-the-fly. So far, we have not explored an on-line approach in J-Orchestra, because of its overheads for regular application execution. Since J-Orchestra has no control over the JVM, these overheads can be expected to be higher than in other systems that explicitly control the runtime environment. Without low-level control, it is hard to keep such overhead to a minimum. Sampling techniques can alleviate the overhead (at the expense of some accuracy) but not eliminate it: some sampling logic has to be executed in each method call, for instance. We hope to explore the on-line profiling direction in the future.

3.3. Rewriting Bytecode for Appletizing

Once all the classes are assigned to their destination sites, J-Orchestra rewrites the application for appletizing, which augments the regular J-Orchestra rewrite with an additional step that modifies unsafe instructions in GUI classes for executing within an applet. The applet security manager imposes many restrictions on what resources applets can access, and many of these restrictions affect GUI code. J-Orchestra inspects the bytecode of an application for problematic operations and "sanitizes" them for safe execution within an applet. Depending on the nature of an unsafe operation, J-Orchestra uses two different replacement approaches. The first approach replaces an unsafe operation with a safe, semantically similar (if not identical) version of it. For example, an unsafe operation that reads a graphical image from disk gets rewritten with a safe operation that reads the same image from the applet's jar file. The second approach, replaces an unsafe operation with a semantically different operation. For example, since applets are not allowed to call `System.exit`, this method call gets replaced with a call to the J-Orchestra runtime service that informs the user that they can exit the applet by directing the web browser to another page. Sometimes, replacing an unsafe instruction requires a creative solution. For example, the applet security manager prevents the `setDefaultCloseOperation` method in class `javax.swing.JFrame` from accepting the value `EXIT_ON_CLOSE`. Since we cannot change the code inside `JFrame`, which is a system class, we modify the caller bytecode to pop the potentially unsafe parameter value off the stack and push the safe value `DO_NOTHING_ON_CLOSE` before calling `setDefaultCloseOperation`. Once unsafe instructions in GUI classes have been replaced, J-Orchestra proceeds with its standard rewrite that ends up packaging all the rewritten classes in client and server jar files ready for deployment.

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

The rewrite also maintains the Swing design invariant of having all event-dispatching and painting code execute in a single event-dispatching thread. Splitting a single-threaded application into a client and server parts creates implicit multithreading. Thus, the server could call client Swing code remotely through RMI on a thread different from the event-dispatching one. To resolve this issue, the rewrite generates special-purpose code inside translator classes. The code uses the existing Swing facility (`SwingUtilities.invokeLater` method) to enable any thread to request that the event-dispatching thread runs certain code.

3.4. Runtime Support for Appletizing

Appletizing works with standard Java-enabled browsers that download the applet code from a remote server. To simplify deployment, the downloaded code is packaged into two separate jar files, one containing the application classes that run on the client and the other J-Orchestra runtime classes. In other words, the client of an appletized application does not need to have pre-installed any J-Orchestra runtime classes, as a Java-enabled browser downloads them along with the applet classes. Once the download completes, the J-Orchestra runtime client establishes an RMI connection with the server and then invokes the main method of the application through reflection. The name of the application class that contains the main method along with the URL of the server's RMI service are supplied as applet parameters in an automatically generated HTML file. This arrangement allows hosting multiple J-Orchestra applets on the same server that can share the same set of runtime classes. In addition, multiple clients can simultaneously run the same applet, but they will also spawn distinct server components. Our approach cannot make an application execute concurrently when it was not designed to do so. In addition to communication, the J-Orchestra applet runtime provides various convenience services such as access to the properties of the server JVM, a capacity for terminating the server process, and a facility for browsing the server's file system efficiently.

4. Case Studies and Discussion

To demonstrate our approach, we appletized three realistic, third-party applications: JBits [10], JNotepad [15], and Jarminator [14]. Our experience confirms the benefits of the approach. Appletizing requires no programming: we did not have to write distribution code or recode the subject applications; it is flexible: each of the subjects has a complex GUI and could not be written as a servlet; it is easy to deploy: all subjects run as applets over a standard browser communicating with a server part; and results in good performance: by putting the GUI code on the client, we transmit less data than transferring all the graphics.

In our measurements, we compare the partitioned applications' behavior to using a remote X display to remotely control and monitor the application. Since all three subjects are interactive applications and we could not modify what they do, 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 appletizing is an even greater win in terms of perceived latency. In all cases, the overall responsiveness of the appletized 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 would not change in a different measurement environment. For reference, however, our environment consisted of a SunBlade 1000 (dual UltraSparc III 750MHz, 2GB RAM) and a Pentium III, 600MHz laptop connected via 10Mbps ethernet.

4.1. JBits

JBits, the largest of the three applications, is an FPGA simulator by Xilinx—a web search shows many instances of industrial use. The JBits GUI (see [10] for a picture of an older version) is very rich with a graphical area presenting the results of the simulation cells, as well as multiple smaller areas presenting the simulated components. The GUI allows connecting to various hardware boards and simulators and depicting them in a graphical form. It also allows stepping through a simulation offering multiple views of a hardware board, each of which can be zoomed in and out, scrolled, etc. The JBits GUI is quite representative of CAD tools in general.

JBits was given to us as a bytecode-only application. 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 anchored by choice on the server. Thus co-anchored objects can 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. With the anchoring by choice, the simulation steps of JBits incur no measurable overhead.

259 classes are always anchored 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). The rest of the classes are anchored on the server site. (We later discuss a variation in which we make some objects mobile.)

The appletized JBits performs 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 3.4MBytes transferred over the network.
- 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 1.8MBytes 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 appletized 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 3.5MBytes of data transferred. Scrolling left once and down once produces

about 2MBytes of data over the network with X, but no network traffic with the J-Orchestra partitioned version. Continuous scrolling over a 10Mbps 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 appletized JBits needs to transfer data over the network, less data get transferred than in the X version. Specifically, the appletized version needs to transfer about 1.28MB of data for a complete simulation step including a redraw of the view. The X window system transfers about 1.68MBytes 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 appletized 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 appletized version can also have its bandwidth requirements optimized by using a version of Java RMI with compression.

Experiment: Mobility. In the previous discussion we did not examine the effects of object mobility. In fact, very few of the potentially mobile objects in JBits actually need to move in an interesting way. The one exception is JBits View Adaptor objects (instances of four `*ViewAdaptor` classes). 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 among 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 2.59MBytes per simulation step (as opposed to 1.28MBytes without a mobility policy). The reason is that the mobile objects are quite large (in the order of 300-400KBytes) 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.

4.2. JNotepad

JNotepad emulates the functionality of the Windows Notepad editor. It allows the user to read and write text files. As in any simple text editor, the functionality of JNotepad consists of a user interface and I/O facilities. The user manipulates the content of a text file through the user interface, which includes the interaction with the I/O facilities for writing and retrieval of files to and from disk. One appletizing scenario for Notepad places the user interface on the client, while processing the I/O on the server.

The analysis for appletizing showed that the application has a total of 106 classes (66 JRE system classes, and 40 application classes). It also assigned 98 classes to the client site, 7 classes to the server site, and left 2 classes unassigned. To help determine a good placement for the unassigned classes named `Center` and `Actions`, we performed a scenario-based profiling that consisted of opening a file, searching for a word in it, changing its content, and saving it back to disk. The data exchange patterns, revealed by the profiling, showed that the `Center` class has been tightly coupled with the client classes, calling each other's methods 17 times. Therefore, the most logical placement for this class is on the client, together with the GUI classes. The `Actions` class exhibited a more complex data exchange pattern, communicating with both the client (18 method calls) and the server (42 method calls). More detailed profiling showed that the data exchange between the server classes and the `Actions` class happens inside the `savE` method, with the rest of the methods communicating only with the client classes. This is exactly a case for which object mobility can provide an elegant solution. The objects of type `Actions` can be created at the client site and then temporarily move to the server for the duration of the `savE` method. As our measurements have shown, this mobility arrangement does not result in less data being transferred over the network, but significantly decreases the number of remote calls made (from 60 to 17).

We compared the behaviors of the partitioned application to the original one, run remotely under the X window system. The test scenario was similar to the profiling one, described above. (We believe that this reflects typical JNotepad use.) The appletized version transferred less than 1/7th the amount of data over the network (~1 MB vs. ~7 MB). With all the GUI operation not generating any network traffic, the appletized version sent data over the network only when reading and writing the text file. Under X, JNotepad, running on the server that had the text file, accessed it directly. However, its every interaction with the GUI resulted in sending data over the network.

4.3. Jarminator

Jarminator is a popular Java application that examines the content of multiple jar files and displays their combined content in a tree view. The user can have only a subset of the content displayed by supplying a wildcard filter. We have appletized Jarminator so that it can examine jar files on a remote machine and display the results locally. The analysis for appletizing showed that the application uses a total of 74 classes: 55 JRE system classes, and 19 application classes. The appletizing analysis assigned 62 classes to the client site, 4 classes to the server sites, and left 8 classes unassigned. A case-based profiling suggested assigning 6 classes to the client, 1 to the server, and did not detect any data exchange with the remaining class. It also did not reveal any communication patterns in which a mobility scenario could be useful.

Again, we compared the behaviors of the partitioned application to the original one, run remotely under the X window system. In this benchmark, we used Jarminator to explore three third-party jar files used by J-Orchestra. The use scenario included loading the jars, navigating through the tree view, and applying wildcard filters to the displayed content. The appletized version exhibits significant benefits, transferring less than 1/30th the amount of data over the network (~500 KB vs. ~15 MB). In fact, operations such as filtering the displayed contents are entirely local in the appletized version and do not generate any network traffic.

4.4. Limitations

Appletizing, just like general application partitioning, is not free of limitations. Applications can be arbitrarily complex and can defy correct partitioning. Furthermore, although we handle common cases of invalid operations inside applets, we do not have an exhaustive approach to sanitize all Java code for applet execution. More common in practice, however, is the case of applications that can be correctly appletized (i.e., they do not employ unsupported Java features such as dynamic loading or code rejected by the applet security manager) yet require manual intervention to override conservative decisions of the J-Orchestra heuristic analyses.

Of our three case studies, JNotepad and Jarminator were partitioned completely automatically within 1-2 hours of time. JBits required more intervention (but still 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 passed by-copy. 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 appletize an application is analogous to that of a system administrator, rather than that of a distributed systems programmer. For instance, in the JBits case we partitioned a 7.5MB 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.

5. Related Work

Several recent systems can be classified as automatic partitioning tools. In the Java world, the closest approaches are the Addistant [23] and Pangaea [20] systems. The Coign system [13] has promoted the idea of automatic partitioning for applications based on COM components. All three systems do not address the problem of partitioning unmodifiable system code (e.g., GUI code) and, thus, are unsuitable for appletizing.

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. Just like appletizing, the Coign approach performs only client-server partitioning. 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 [13] 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 in a general purpose language (like Java, C, C#, or C++) where pointers are prevalent, unless a strict component-based implementation methodology is followed.

JavaParty [11][19] 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. The implementation techniques used, however, are very similar to J-Orchestra, especially for the newest versions of JavaParty [11]. Similar comments apply to the FarGo [12] and AdJava [8] systems. Notably, however, FarGo has focused on grouping classes together and moving them as a group. FarGo groups are similar to J-Orchestra anchored groups. In fact, groups of J-Orchestra objects that are all anchored by choice could well move, as long as they do it all together. We have not investigated such mobile groups, however.

Automatic partitioning is essentially a Distributed Shared Memory (DSM) technique. Nevertheless, automatic partitioning differs from traditional DSMs in several ways. First, automatic partitioning systems do not change the runtime system, but only the application. This is essential for deploying applets that will work on standard VMs inside web browsers. Traditional DSM systems like Munin [5], Orca [2], and, in the Java world, cJVM [1], and Java/DSM [24] 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 unmodifiable system code, as discussed). Among distributed shared memory systems, the ones most closely resembling the J-Orchestra approach are object-based DSMs, like Orca [2].

Mobile object systems, like Emerald [3] have formed the inspiration for many of the J-Orchestra ideas on object mobility scenarios.

Both the D [17] and the Doorastha [7] 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. All the burden is shifted to the programmer to specify what semantics is valid for a specific class (e.g., whether objects are mobile, whether they can be passed by-copy, etc.). Programming in this way requires full understanding of the application behavior and can be error-prone: a slight error in an annotation may cause insidious inconsistency errors.

6. Conclusions

Adding distributed capabilities to existing programs is currently one of the most important software evolution tasks in practice [16]. We presented appletizing, a semi-automatic approach to transforming a Java GUI application into a client-server application. We discussed the motivation, benefits, and J-Orchestra support for appletizing, and validated our approach via a set of case studies and associated benchmarks. We believe that our approach, having the benefits of automation, flexibility, ease of deployment, and good performance, is a useful tool for software evolution, and that similar tools will become mainstream in the future.

Acknowledgements. This research was supported by the NSF under Grants No. CCR-0220248 and CCR-0238289.

References

[1] Yariv Aridor, Michael Factor, and Avi Teperman, "CJVM: a Single System Image of a JVM on a Cluster", in Proc. *ICPP'99*.
[2] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Ruhl, and M. Frans Kaashoek, "Performance Evaluation of the Orca Shared-Object System", *ACM Trans. on Computer Systems*, 16(1):1-40, February 1998.
[3] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, "Distribution and Abstract Types in Emerald", in *IEEE Trans. Softw. Eng.*, 13(1):65-76, 1987.
[4] Jon Byous, "Opportunities Everywhere", `http://java.sun.com/javaone/general_sessions1.html`.
[5] John B. Carter, John K. Bennett, and Willy Zwaenepoel, "Implementation and performance of Munin", *13th ACM Symposium on Operating Systems Principles* (SOSP), 1991.
[6] Markus Dahm, "Byte Code Engineering", *JIT* 1999.
[7] Markus Dahm, "Doorastha—a step towards distribution transparency", *JIT* 2000.
[8] Mohammad M. Fuad and Michael J. Oudshoorn, "AdJava— Automatic Distribution of Java Applications", 25th *Australasian Computer Science Conference (ACSC)*, 2002.
[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, "The Java Language Specification", Second Edition, Addison Wesley, 2000.
[10] Steven A. Guccione, Delon Levi and Prasanna Sundararajan, "JBits: A Java-based Interface for Reconfigurable Computing", *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.
[11] Bernhard Haumacher, Jürgen Reuter, Michael Philippsen, "JavaParty: A distributed companion to Java", `http://wwwipd.ira.uka.de/JavaParty/`
[12] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit, "Dynamic Layout of Distributed Applications in FarGo", *Int. Conf. on Softw. Engineering (ICSE),* 1999.
[13] Galen C. Hunt, and Michael L. Scott, "The Coign Automatic Distributed Partitioning System", *3rd Symposium on Operating System Design and Implementation (OSDI)*, 1999.
[14] Jarminator: Free software application. From `http://www.javasvet.net/prj/jarminator/`
[15] JNotepad: Free software application. From `http://www.pscode.com/`
[16] Nelson King, "Partitioning Applications", *DBMS and Internet Systems* magazine, May 1997. See `http://www.dbms-mag.com/9705d13.html` .
[17] Cristina Videira Lopes and Gregor Kiczales, "D: A Language Framework for Distributed Programming", PARC Technical report, February 97, SPL97-010 P9710047.
[18] Alan Messer, Ira Greenberg, Philippe Bernadat, Dejan Milojicic, Deqing Chen, T.J. Giuli, Xiaohui Gu, "Towards a Distributed Platform for Resource-Constrained Devices", *International Conference on Distributed Computing Systems (ICDCS)*, 2002.
[19] Michael Philippsen and Matthias Zenger, "JavaParty - Transparent Remote Objects in Java", *Concurrency: Practice and Experience*, 9(11):1125-1242, 1997.
[20] Andre Spiegel, *Automatic Distribution of Object-Oriented Programs*, PhD Thesis. FU Berlin, FB Mathematik und Informatik, December 2002.
[21] Eli Tilevich and Yannis Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning", *European Conference on Object-Oriented Programming (ECOOP)*, June 2002.
[22] Eli Tilevich and Yannis Smaragdakis, "Portable and Efficient Distributed Threads for Java", *Middleware'04* conference, October 2004.
[23] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano, "A Bytecode Translator for Distributed Execution of 'Legacy' Java Software", *European Conference on Object-Oriented Programming (ECOOP)*, June 2001.
[24] Weimin Yu, and Alan Cox, "Java/DSM: A Platform for Heterogeneous Computing", *Concurrency: Practice and Experience*, 9(11):1213-1224, 1997.