CHAPTER V

APPLICABILITY AND CASE STUDIES

This chapter argues that this dissertation explores algorithms, techniques, and tools for separating distribution concern that can be a valuable addition to the working programmer's toolset. The content of this chapter, as its title suggests, is divided into two parts: discussing general applicability issues followed by supporting our claims through a series of case studies. The discussion starts by showing how NRMI, GOTECH, and J-Orchestra compare with each other and how they follow the translucent approach to separating distribution concerns. Then we take a closer look at each software tool from the applicability perspective. This includes identifying for each tool the distribution concerns that it separates, the common architectural characteristics of the centralized applications that it can effectively and efficiently transform for distributed execution, and the programming scenarios under which programmers are most likely to find it useful. The second part of the chapter presents a series of case studies that apply each of the three tools to various programming scenarios, thus supporting our applicability claims empirically.

5.1 Applicability of the Translucent Approach

This dissertation explores software tools that separate distribution concerns by following the approach we call "translucent." This approach, while aiming at distribution transparency, nevertheless, does not attempt to mask all the differences between the cen-

tralized and distributed execution models. NRMI—a middleware mechanism, GOTECH—a code generator, and J-Orchestra an automatic partitioning system demonstrate the value of the translucent approach in their own categories. On the scale of automation, NRMI is the least automatic tool, being just a middleware mechanism, GOTECH introduces distribution into centralized programs, given the programmer's annotations, and J-Orchestra automates the entire distribution process, requiring only high-level GUI-based input from the programmer. Next we describe in greater detail how each of these tools follows the translucent approach.

NRMI makes remote calls look like local calls as far as the parameters passing semantics is concerned for stateless servers and single-threaded clients. It is the programmer's responsibility to ensure that these preconditions hold true. At the same time, NRMI does not hide the possibility of partial failures, and similarly to regular RMI, every remote call can throw various remote exceptions, and the programmer is responsible for supplying custom code for catching and handling them. Thus, with NRMI and call-by-copy-restore, remote calls have a closer semantics to the one of local calls without trying to hide the possibility of partial failure.

GOTECH uses NRMI as its building block, but has more preconditions for successful application. Prior to specifying which local calls GOTECH should transform into remote calls, the programmer has to be aware that the structure of the original application is amenable to such a transformation. This includes not only the preconditions for the successful application of NRMI (i.e., a stateless server and a single-thread client) but also that the centralized program follows the strict client-server communication model. The programmer is also responsible for providing custom code for handling the remote exceptions

that could be raised during every remote method invocation. Thus, GOTECH relieves the programmer from having to write tedious and error-prone distribution code by hand, but the code that it generates does not attempt to hide the fact that the distribution has taken place.

Finally, J-Orchestra is the most automatic of the tools and also separates the largest number of distribution concerns. Nevertheless, J-Orchestra is not a distributed shared memory system and aims at functional distribution, putting the code near the resources it manages. Despite having some preconditions for successful application that are discussed in Section 5.4, J-Orchestra works correctly with a very broad subset of the Java language and shares none of the preconditions of NRMI and GOTECH. Specifically, a distributed program, created through J-Orchestra partitioning, can have stateful servers and multi-threaded clients. As far as the possibility of failures is concerned, J-Orchestra uses regular Java RMI, and a sophisticated user can provide custom error handling in response to raised remote exceptions. Thus, J-Orchestra relieves the programmers from having to figure out all the complex data sharing scenarios done through references, allowing them to concentrate on the challenging issues in distributed computing such as handling partial failures.

Next we take a closer look at the applicability issues of NRMI, GOTECH, and J-Orchestra in turn.

5.2 Applicability of NRMI: Usability Call-by-Copy-Restore vs. Call-by-Copy

Compared to call-by-copy, call-by-copy-restore semantics offers better usability, for it simulates the local execution semantics very closely, as discussed in Section 2.4.1.

Clearly, call-by-copy-restore semantics can be achieved by using call-by-copy and adding application-specific code to register and re-perform any updates necessary. Nevertheless, taking this approach has several disadvantages:

- The programmer has to be aware of all aliases in order be able to update the values changed during the remote call, even if the changes are to data that became unreachable from the original parameters.

- The programmer needs to write extra code to perform the update. This code can be long for complex updates (e.g., up to 100 lines per remote call for the microbenchmarks we discuss in Section 5.5.3).

- The programmer cannot perform the updates without full knowledge of what the server code changed. That is, the changes to the data have to be part of the protocol between the server programmer and the client programmer. This complicates the remote interfaces and specifications.

As we discussed in Section 2.2 on page 15, a call-by-copy-restore semantics is most valuable in the presence of aliased data. Aliasing occurs as a result of several common implementation techniques in mainstream programming languages. All of these techniques produce code that is more convenient to write using call-by-copy-restore middleware than using call-by-copy middleware. Specific examples include:

- *Common GUI patterns such as model-view-controller*. Most GUI toolkits register multiple views, all of which correspond to a single model object. That is, all views alias the same model object. An update to the model should result in an update to all of the views. Such an update could be the result of a remote call.

- A variant of this pattern occurs when GUI elements (e.g., menus, toolbars) hold aliases to program data that can be modified. The reason for multiple aliasing is that the same

data may be visible in multiple toolbars, menus, and so forth or that the data may need to be modified programmatically with the changes reflected in the menu or toolbar. For example, we distribute with NRMI a modified version of one of the Swing API example applications. We changed the application to be able to display its text strings in multiple languages. The change of language is performed by calling a remote translation server when the user chooses a different language from a drop-down box. (That is, the remote call is made in the event dispatching thread, conforming to Swing thread programming conventions [81].) The remote server accepts a vector of words (strings) used throughout the graphical interface of the application and translates them between English, German, and French. The updated list is restored on the client site transparently, and the GUI is updated to show the translated words in its menus, labels, and so on. The distributed version code (using the RMI drop-in replacement implementation) has only two tiny changes compared to local code: a single class needs to implement the NRMI marker interface `java.rmi.Restorable`, and a method has to be looked up using a remote lookup mechanism before getting called. In contrast, the version of the application that uses regular Java RMI has to use a more complex remote interface for getting back the changed data and the programmer has to write code in order to perform the update.

- *Multiple indexing*. Most applications in imperative programming languages create some multiple indexing scheme for their data. For example, a business application may keep a list of the most recent transactions performed. Each transaction, however, is likely to also be retrievable through a reference stored in a customer's record through a reference from the daily tax record object, and so forth. Similarly, every customer may be retrievable from a data structure ordered by zip code and from a second data structure ordered by name. All of these references are aliases to the same data (i.e., customers, business transactions). NRMI allows such references to be updated correctly as a result of a remote call (e.g., an update of purchase records from a different location or a retrieval of a customer's address from a central database), in much the same way as they would be updated if the call were local.

5.3 Applicability of GOTECH: What are the Distribution Concerns and Can They Be Separated?

For insight into identifying "distribution concerns," we refer to the "differences" between local and distributed models of computation listed in Waldo et al.'s well-known "A Note on Distributed Computing" [96]. The distribution concerns that GOTECH aims to separate fall into three main groups: semantics, performance, and conventions.

5.3.1 Semantics

Consider a centralized application written in a modular fashion with separate objects handling distinct parts of the functionality of the application. It might seem that moving a part of the functionality to a remote machine is just a matter of making some object remotely accessible by the rest of the application. Nevertheless, objects can be sharing data through memory references (which are valid only in a single address space). Of course, one could emulate a single address space over a network of nodes by making all references be over the network. However, such an emulation would be prohibitively slow.

As a result, the semantics of remote method calls are different from the semantics of local calls under standard middleware. That is, the same code will behave differently if executed in the same process and if executed as a remote call (using CORBA, RMI, DCOM, and so forth) on a different machine. Therefore, the lack of a shared address space is the single most important conceptual difference introduced by distribution. This problem cannot be solved in a fully general way. For instance, an application may have a structure such that all its parts are tightly coupled, access each other's data (or OS-level resources, like I/O) directly and depend on reading the latest values of these data. In this case, efficient

distribution is impossible without a change in the application structure. Therefore, the assumption of the GOTECH approach is that the application is amenable to added distribution without a fundamental change in the application structure. In this case, the memory semantics issue can be alleviated if the programmer has control over the calling semantics and if local semantics can be emulated under certain assumptions so that the programmer does not need to write a lot of tedious code. Distribution also requires changes to the client of a remote object to become aware of the possibility of partial failures. Again, this problem cannot be solved in a fully general way, but Java language designers used the exception mechanism to ensure partial failure awareness: the client of a remote call needs to handle various exceptions that might arise in response to various partial failure conditions.

5.3.2 Performance

With processor speed continuing to increase at a much higher rate than network performance, remote calls have become more costly than ever compared to local calls. When some local calls suddenly become remote, the resulting distributed application may become unusable due to a slowdown by orders of magnitude. When applying distribution as a separate step, one has to be aware of such latencies when deciding whether an object can be moved to a remote site. An object can be moved to a remote site only if it is not tightly coupled with the rest of the application. For this reason, the programmer should have complete control over the location of objects.

5.3.3 Conventions

Using a middleware mechanism such as RMI, CORBA, DCOM, and so forth to enable distribution has become a common business practice. Since GOTECH aims to

remove the low-level technical barriers to aspectizing distribution, our main challenge is to change application code so that it interfaces with distribution middleware, which entails manipulating code according to established conventions. In object-oriented distributed systems, types are often used to mark an object that can interact with the middleware runtime services. For example, to interact with such a runtime service an object might have to implement certain interfaces by providing methods that are called by the middleware at runtime. Another example would be to declare the remote methods of an object as throwing exceptions for potential network errors. The client code requires some modifications as well. A call to a remote object constructor might have to be replaced by a sequence of calls to a registry service. Making such changes can be quite tedious. Tool vendors have made some inroads in alleviating the task of converting plain objects so that they conform to a given framework convention. One such example is Microsoft's Class Wizard for Visual C++, which creates an MFC class from a given COM object. However, none of these industrial tools help the programmer make changes to the *clients* of the modified object.

The authors of the “Note” suggest that, “providing developers with tools that help manage the complexity of handling the problems of distributed application development as opposed to the generic application development is an area that has been poorly addressed.” Hopefully, the ideas introduced by GOTECH can help in providing developers with such tools.

5.4 Applicability of J-Orchestra: Conditions for Successful Automatic Partitioning

"It's not how well the bear dances;
it's that it can dance at all"

J-Orchestra can handle a large subset of Java and, thus, can correctly partition a large class of realistic unsuspecting applications. Nevertheless, among these, J-Orchestra will be useful only in a few well-defined cases. Automatic partitioning is not a substitute for general distributed systems development. The striking element of the approach is not that it is widely applicable but that it is at all applicable, given how automated it is.

We introduce the term *embarrassingly loosely coupled* to describe the kinds of applications to which J-Orchestra is applicable. An embarrassingly loosely coupled application satisfies two criteria:

- it has components that exchange little data with the rest of the application, and
- these components are statically identifiable by looking at the structure of the application code at the class or the module level.

That is, by looking at static relations 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 distinct components should be very small. In other words, an application should have very clear communication and locality patterns. Since the application logic will remain the same, a large number of remote accesses will be detrimental to performance. This requirement stems from the intrinsic difference in latency between local and remote method calls. As an

illustration, consider a sequence of method calls constituting an execution scenario. In a centralized, monolithic application, an execution scenario comprises local method calls, executing in a shared address space. When partitioning takes place, in the same execution scenario some of the method calls become remote, invoked through an RPC mechanism such as Java RMI. As the latency of a remote call is several orders of magnitude larger than that of a local call, a partitioned version of a sequential centralized application is expected to take longer to execute.[1]

The slowdown factor can be defined as the difference in total execution time (disregarding such additional factors as waiting for user input or interacting with the OS) between the original centralized application and its partitioned version. Of course, applications differ in terms of how much slowing down they can afford as a result of partitioning before the process would render them unusable. Nevertheless, a performance optimization, aimed at reducing the total number of remote calls, would be beneficial for any application. What determines the total number of remote calls made by a particular partitioned application is not just the initial static class placement but also various object mobility scenarios. Many modern networks, in which latency is more of an issue than bandwidth, present optimization opportunities via the means of object mobility (i.e., moving method arguments to or the return value from the site of a remote call) that could reduce the total number of remote calls. Approaches to estimating the expected slowdown factor of an application include online or offline profiling that could take into consideration both the initial static placement of objects and object mobility scenarios. Once presented with the profiling results, the pro-

1. An additional slowdown results from the fact that a partitioned version of a centralized application always ends up making more (local) method calls in a given execution scenario due to such mechanisms as proxy indirection and object factory lookup introduced by the partitioning.

grammer can decide whether the expected slowdown factor is acceptable for a given application.

Once partitioned, an embarrassingly loosely coupled application must not share objects among partitions that are used by unmodifiable code (e.g., OS or JVM code) and should have synchronous communication patterns. If either of these two properties do not hold true or if good performance or reliability requires asynchronous communication, the application structure needs to change.

Hence, 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). Several special-purpose technologies do this already: distributed file systems allow storage on remote disks; remote desktop applications (e.g., VNC [69], X [70]) allow transferring graphical data from a remote machine; network printer protocols let users print remotely. Nevertheless, the advantage of automatic partitioning is that it can put the code near the resource that it controls. For instance, if a graphical representation can be computed from less data than it takes to transfer the entire graphical representation over the network, then J-Orchestra has an advantage. Some mainstream technologies put code near a resource such as Java applets, which move graphics-producing code from a server to a client with the screen on which the graphics will be displayed. However, this solution is inflexible, as it requires the entire program to move across the network. In contrast, applet-

izing (Section 4.7), a specialization of automatic partitioning, can split an application so that any part of it can become a “virtual applet” and can run on a client machine.

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.

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

5.5 NRMI Case Studies

Before presenting the results of NRMI performance experiments, we describe the performance optimizations that we applied to the RMI replacement implementation of NRMI. These optimizations demonstrate that *copy-restore middleware can be optimized for real-world use*, which is the main point of our experiments. Although NRMI emphasizes usability, its implementation can be quite efficient: the RMI replacement implementation is optimized and suffers only small overheads. The optimized NRMI for JDK 1.4 is about 20% slower than regular RMI for JDK 1.4. To put this number in perspective, this also means that NRMI for JDK 1.4 is about 20-30% faster than regular RMI for the previous version, JDK 1.3.

5.5.1 NRMI Low-Level Optimizations

In principle, the only significant overhead of call-by-copy-restore middleware over call-by-copy middleware is the cost of transferring the data back to the client after the remote routine execution. In practice, middleware implementations suffer several overheads related to processing the data, so that processing time often becomes as significant as network transfer time. Java RMI has been particularly criticized for inefficiencies, as it is implemented as a high level layer on top of several general purpose (and thus slow) facilities—RMI often has to suffer the overheads of security checks, Java serialization, indirect access through mechanisms offered by the Java Virtual Machine, and so forth. NRMI has to suffer the same overheads to an even greater extent, since it has to perform an extra traversal and copying over object structures.

Our implementation of NRMI as a full replacement of Java RMI has two versions: a "portable", high-level one and an "optimized" one. The *portable* version makes use of high-level features such as Java reflection for traversing and copying object structures. Although NRMI is currently tied to Sun's JDK, the portable version works with JDK 1.3, 1.4, and 1.5 on all supported platforms. The portability means some loss of performance: Java reflection is a very slow way to examine and update unknown objects. Nevertheless, our NRMI implementation minimizes the overhead by caching reflection information aggressively. Additionally, the portable version uses native code for reading and updating object fields without suffering the penalty of a security check for every field. These two optimizations give a >200% speedup to the portable version, but still do not achieve the optimized version's performance.

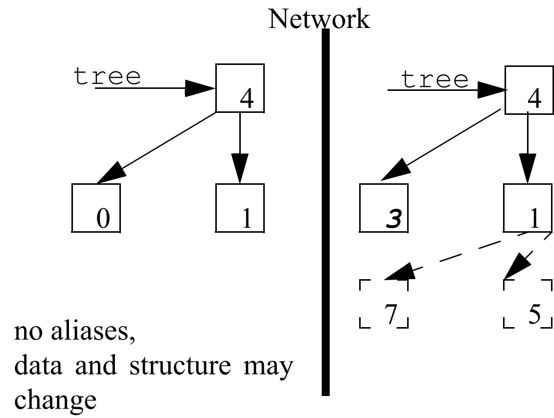
The *optimized* version of NRMI only works with JDK 1.4 and 1.5 and takes advantage of special features exported by the JVM in order to achieve better performance. The performance of regular Java RMI improved significantly between versions 1.3 and 1.4 of the JDK (as we show in our measurements). The main reason was the flattening of the layers of abstraction in the implementation. Specifically, object serialization was optimized through non-portable direct access to objects in memory through an "Unsafe" class exported by the Java Virtual Machine. The optimized version of NRMI also uses this facility to quickly inspect and change objects.

5.5.2 Description of Experiments

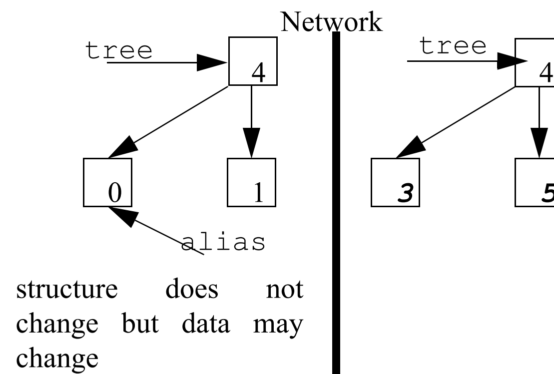
In order to see how our implementation of call-by-copy-restore measures up against the standard implementation of RMI, we created three micro-benchmarks. Each benchmark consists of a single randomly-generated binary tree parameter passed to a remote method. The remote method performs random changes to its input tree. *The invariant maintained is that all the changes are visible to the caller*. In other words, the resulting execution semantics is as if both the caller and the callee were executing within the same address space. With NRMI or distributed call-by-reference (through remote pointers, as in Figure 2-3) this is done automatically. For call-by-copy, the programmer needs to simulate it by hand.

We have considered three different scenarios of parameter use, listed in the order of difficulty of achieving the call-by-copy-restore semantics "by-hand" using the means provided by RMI.

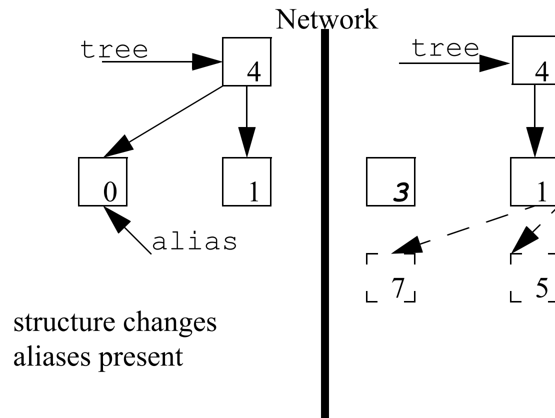
• In the first benchmark scenario, we assume that none of the objects reachable from the parameter is aliased by the client.



• In the second benchmark scenario, we allow aliases but assume that the structure of the tree stays the same (although the tree data may be modified by the remote method).



• In the third benchmark scenario, aliasing and modifications can be arbitrarily complex: tree nodes can be aliased on the client site and the tree structure can be changed by the remote call.



Consider how a programmer can replay the server changes on the client using regular Java RMI in each of the three cases. We assume that the programmer is fully aware of the server's behavior, as well as of whether aliases exist on the client site.

• In the first case, the parameter just has to be returned as the return value of the remote method. Once the remote call completes, the reference pointing to the original data structure gets reassigned to point to the return value. This will work for any changes to both the data and structure of the tree. The only complication here is that the method might already have a return value. Resolving this problem would require defining a special return class type that would contain both the original return type and the parameter. Besides the code for this new return class type itself, some additional code has to be written to call its constructor, populate it with its constituent members on the callee site, and retrieve them when the call completes.

• In the second case, the client needs to reassign the aliases pointing to some nodes in the original tree to point to the corresponding nodes in the new tree. After this step is performed, the reference reassignment described in the previous benchmark can be used. If

the programmer knows all the aliases, as well as where in the tree they point to (i.e., how to get to the aliased node from the tree root) then the aliases can be reassigned directly. If, however, the programmer only knows the aliases but not how to get to the aliased nodes, then a search needs to be performed before the update takes place. Both the original and the modified trees (which are now isomorphic) can be traversed simultaneously. Upon encountering each node, all aliases should be reassigned from pointing from the original tree to the modified one.

- In the third case, returning the changed structure alone is not very useful since the original and the modified trees are no longer isomorphic. To complicate matters further, the remote method invocation might make some changes to some of the tree nodes' data that were aliased by the caller and then disconnect them from the tree structure. This way the modified data nodes might no longer be part of the tree structure. Obviously just returning the new tree is not enough. Emulating the call-by-copy-restore or call-by-reference semantics is particularly cumbersome in this case. The simplest way to do it is by having the remote method create a "shadow tree" of its tree parameter prior to making any changes to it. The "shadow tree" points to the original tree's data and serves as a reminder of the structure of the original tree. Then both the parameter tree and the "shadow tree" are returned to the caller. The "shadow tree" is isomorphic to the original parameter and can be used for simultaneous traversal and copying of aliases. After that the new tree is used for the reference reassignment operation as in the previous cases. Note that correct update is not possible without modifying both the server and the client.

For all benchmarks, the NRMI version of the distributed code is quite similar to the local version, with the exception of remote method lookup and declaring a class to be `Restorable`. Analogous changes have to be made in order to go from the local version to the distributed one that updates client data correctly using regular Java RMI. Several extra lines of code have to be added/modified in the latter case, however. For all three benchmarks, about 45 lines of code were needed in order to define return types. For the second

and third benchmark scenario, an extra 16 lines of code were needed to perform the updating traversal. For the third benchmark scenario, about 35 more lines of code were needed for the "shadow tree".

5.5.3 Experimental Results

For each of these benchmarks, we measure the performance of call-by-copy (RMI), call-by-copy-restore (NRMI), and call-by-reference implemented using remote pointers (RMI). (Of course, NRMI can also be used just like regular RMI with identical performance. In this section when we talk of "NRMI" we mean "under call-by-copy-restore semantics".) For reference, we also provide three base line numbers by showing how long it takes to execute the same methods within the same JVM locally, on different JVMs through RMI but on the same physical machine, and on different machines but without caring to restore the changes to the client (i.e., only sending the tree to the server but not sending the changed tree back to the client). We show measurements for both versions of NRMI and both JDK 1.3 and JDK 1.4. The environment consists of a SunBlade 1000 (two UltraSparc III 750MHz processors and 2GB of RAM) and a Sun Ultra 10 (UltraSparc II 440MHz) machines connected with a 100Mbps effective bandwidth network. This environment certainly does not unfairly benefit NRMI measurements—the network speed is representative of networks in which high-level middleware is used and the machines are on the low end of today's performance spectrum. For faster machines and slower networks, the performance of NRMI would strictly improve relative to the baselines.

The results of our experiments are shown in Table 5-1 to Table 5-6. All numbers are in milliseconds per remote call, rounded to the nearest millisecond. We ensured (by "warm-

ing" the JVM) that all measured programs had been dynamically compiled by the JVM before the measurements.

Table 5-1. Baseline 1—Local Execution (processing overhead) on both the fast (750MHz) and the slow (440MHz) machine.

Bench mark/ Tree Size	**JDK 1.3**				**JDK 1.4**			
	16	**64**	**256**	**1024**	**16**	**64**	**256**	**1024**
I	<1 / <1	<1 / 1	1 / 2	6 / 8	<1 / <1	<1 / 1	1 / 1	4 / 6
II	<1 / 1	1 / 1	4 / 5	15 / 20	<1 / 1	1 / 1	3 / 4	12 / 16
III	<1 / 1	1 / 2	5 / 6	19 / 24	<1 / 1	1 / 1	4 / 5	15 / 19

Table 5-2. Baseline 2—RMI Execution, without Restore (one-way traffic).

Bench mark/ Tree Size	**JDK 1.3**				**JDK 1.4**			
	16	**64**	**256**	**1024**	**16**	**64**	**256**	**1024**
I	3	7	18	65	2	4	9	33
II	3	7	21	74	3	4	12	41
III	3	8	22	79	3	5	12	44

Table 5-3. Baseline 3—RMI Execution with Restore on local machine (no network overhead).

Bench mark/ Tree Size	**JDK 1.3**				**JDK 1.4**			
	16	**64**	**256**	**1024**	**16**	**64**	**256**	**1024**
I	3	7	17	59	3	4	11	41
II	4	8	19	67	3	5	13	48
III	4	9	24	87	3	6	16	66

Table 5-4. RMI Execution with Restore (two-way traffic).

Bench mark/ Tree Size	**JDK 1.3**				**JDK 1.4**			
	16	**64**	**256**	**1024**	**16**	**64**	**256**	**1024**
I	5	11	29	102	4	6	18	68
II	5	12	32	112	4	7	21	77

Bench mark/ Tree Size	JDK 1.3				JDK 1.4			
	16	64	256	1024	16	64	256	1024
III	6	13	38	143	4	9	27	106

Table 5-5. NRMI (Call-by-copy-restore). Both the portable and optimized version shown for JDK 1.4.

Bench mark/ Tree Size	JDK 1.3				JDK 1.4			
	16	64	256	1024	16	64	256	1024
I	6	13	36	130	5 / 4	8 / 8	25 / 22	93 / 82
II	6	13	38	141	5 / 4	9 / 8	27 / 24	103 / 95
III	6	14	39	146	5 / 4	9 / 8	28 / 25	106 / 97

Table 5-6. Call-by-Reference with Remote References (RMI).

Bench mark/ Tree Size	JDK 1.3				JDK 1.4			
	16	64	256	1024	16	64	256	1024
I	41	50	87	-	44	48	124	-
II	35	50	85	-	49	53	95	-
III	113	123	164	-	131	131	228	-

The local measurements of Table 5-1 are given for both the fast and the slow machines. The local measurements of Table 5-6 are from the dual processor SunBlade machine. (This allowed us to avoid context switching and get a fair measurement. The numbers were significantly tainted on a uniprocessor machine.)

The main observations from these measurements are as follows:

- The benchmarks have very low computation times—their execution consists mostly of middleware processing and data transmission.

- Java RMI in JDK 1.4 is significantly faster than RMI in JDK 1.3. The speedup is in the order of 50-60% for this experimental setting. The speedup will be lower for a network that is slower relative to the processor speeds.

- The results of Table 5-4 minus the corresponding results of Table 5-3 will only yield an upper bound for the raw data transmission time, because the Table 5-3 results were computed exclusively on the fast (750MHz) machine while the Table 5-4 results include computation on both the fast and the slow (440MHz) node. The difference between the raw data transmission time and the "Table 5-4-minus-Table 5-3" value can be as high as the difference between the computation times on the fast and slow machines, shown in Table 5-1. Even then, however, JDK 1.3 seems to perform much better when no network is involved than the corresponding difference in JDK 1.4. This leads us to conclude that probably RMI in JDK 1.4 uses the underlying OS/networking facilities much more efficiently than JDK 1.3 and this difference disappears when the two hosts are sharing memory. We independently corroborated the raw data transmission time shown in the tables by profiling the benchmarks and noting the amount of time they spent blocked for I/O. We found that the real transmission time for JDK 1.3 is much higher even for transmitting the exact same amount of data as 1.4.

- For benchmarks I and II, NRMI is quite efficient. Even the portable version is rarely more than 30% slower than the corresponding RMI version. The optimized implementation of NRMI is about 20% slower than RMI in JDK 1.4. This is certainly fast enough for use in real applications. For instance, the optimized implementation of NRMI for JDK 1.4 is 20-30% faster than regular RMI in JDK 1.3.

- For benchmark III, which is hard to simulate by hand with call-by-copy alone, the portable implementation of NRMI gets similar performance to regular RMI in all cases, while the optimized implementation is faster. The cause is *not* the processing time for restoring the values changed by the header. (In fact, we performed the same experiments ignoring the manual restoring of changes and got almost identical timings.) Instead, the reason is that the regular RMI version transfers more data: the "shadow tree" is a simple way to

emulate the local semantics by hand, but stores more information than that of the NRMI linear map. (Specifically, it stores all the original structure of the tree instead of just pointers to all the reachable nodes.) The only alternative would be to compute a linear mapping to all reachable nodes on both sides, effectively imitating NRMI in user space.

- Call-by-reference implemented by remote pointers is extremely inefficient (as expected). Every access to parameter data by the remote method results in network traffic. Java RMI does not seem fit for this kind of communication at all—the memory consumption of the benchmarks grew uncontrollably. For the 1,024 node trees, the benchmarks took more than 600ms per case (repeated over 1,000 times) and in fact failed to complete as they exceeded the 1GB heap limit that we had set for our Java virtual machines. The reason for the memory leak is that the references back from the server to the client create distributed circular garbage. Since RMI only supports reference counting garbage collection, it cannot reclaim the garbage data.

The conclusion from our experiments is that NRMI is optimized enough for real use. NRMI (copy-restore) for JDK 1.4 is close to the optimized RMI in JDK 1.4 and faster than regular RMI (call-by-copy with results passed back) in JDK 1.3. Of course, with NRMI the programmer maintains the ability to use call-by-copy semantics when deemed necessary. When, however, a more natural programming model is desired, NRMI is without competition—the only alternative is the very slow call-by-reference through RMI remote pointers.

5.6 The GOTECH Case Study

In this section we present an example of applying the GOTECH framework to convert a scientific application into a distributed application interacting with an application server. The original application is a thermal plate simulator. Its back-end engine performs

the CPU-intensive computations and its front-end GUI visualizes the results. (The back-end engine can also be configured to receive input from real heat sensors.)

The distribution scenario we want to accomplish is to separate the back-end simulation functionality from the rest of the application. and to place it on a powerful remote server machine. There are several benefits gained by this distribution scheme. First, it takes advantage of the superior computing power of a remote server machine. Second, multiple clients can share the same simulation server. Finally, if real heat sensors are used, the user does not have to be in the same physical location with the sensors to run the experiment.

The kind of distribution we examine is very similar to the distribution scenario of the Health Watcher application by Soares et al. [72]. (We sought to replicate the experiment of Soares et al. and re-engineer the Health Watcher system, but unfortunately the code is proprietary and was not made available to us.) The distribution scenario for Health Watcher was one where the GUI was running remotely from the core application and used a facade class to communicate with it. Near-identical issues are raised with our thermal plate simulator. Note, however, that, unlike Soares et al., we concentrate only on distribution and do not concern ourselves with persistence aspects.

A simplified UML diagram for the original version of the thermal plate simulator is shown in Figure 5-1. We have laid out the class diagram so that the front-end and back-end are clearly visible. The hierarchy under interface `Plate` contains the types of the objects that form the connecting link between the application's front-end and back-end. The graphical front-end creates a `Plate` object and several visual component objects reference it and query it to obtain the necessary data when performing their drawing operations. The `Plate`

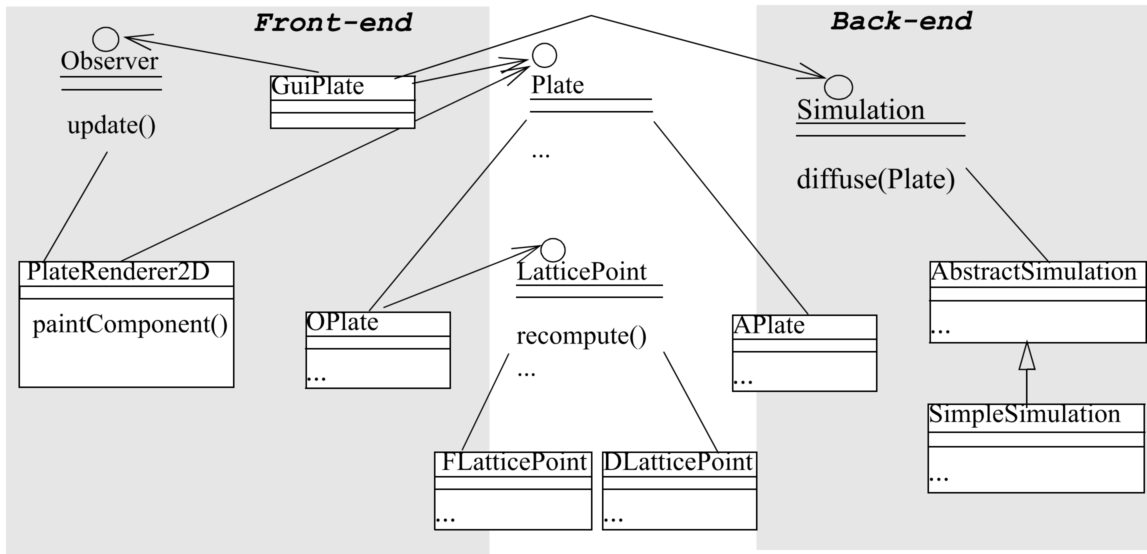


Figure 5-1: UML class diagram of the Thermal Plate Simulator functionality

object gets modified by being sent as a parameter to the `diffuse` method in the `Simulation` class. Once the `diffuse` method returns having modified its `Plate` parameter, the front-end is signaled to repaint itself. The visual components can access the updated data of the `Plate` object and redraw. Note that the main computation logic of the thermal plate simulation is not distributed—the results are the only data transferred over the network for remote display and simulation control.

Accomplishing the outlined distribution takes two steps:

- Converting simulation classes into EJBs and deploying them in an application server.
- Changing the client code to interact with an application server and EJBs instead of plain Java objects.

Notice that making simulation classes remote while preserving the original execution semantics requires special handling for remote method parameters. The `Plate` object that participates in a complicated aliasing (i.e. multiple referencing) scenario now becomes

a parameter of a remote call to an EJB. If a copy-restore mechanism is not provided by the application server, then the process of bridging the differences between local (by-reference) and remote (by-copy) parameter passing semantics becomes a tedious and complicated task. The use of NRMI (copy-restore semantics) completely eliminates the need for special purpose code to reproduce the back-end changes to the `Plate` object inside the front-end.

In-order for GOTECH to perform the required changes, we add some XDoclet-specific tags. Below are all the tags that are needed to convert a plain class `lattice.SimpleSimulation` into a stateless session Enterprise Java Bean.

```
/**
 * @ejb:bean name="SimpleSimulation"
 *           display-name="SimpleSimulation Bean"
 *           type="Stateless"
 *           transaction-type="Container"
 *           jndi-name="ejb/test/SSim"
 */

package lattice;
class SimpleSimulation {
...
/**
 * @ejb:interface-method view-type="remote"
 * @jboss:method-parameters copy-restore="true"
 */
 public void diffuse (Plate plate) { ... }
...
}
```

The tags entered in `lattice.SimpleSimulation` will convert the class into an EJB and will also change all its clients consistently. XDoclet generates the home and remote interface, as well as the bean class, all derived from the original source code for

`SimpleSimulation`. For example, the generated code for the home interface of the `Sim-pleSimulation` EJB (slightly simplified) is:

```
package simulations;
// [Redundant import statements removed]
/**
 * Home interface for SimpleSimulation.
 * @xdoclet-generated at [date] [time]
 */

public interface SimpleSimulationHome
  extends javax.ejb.EJBHome
{
 public static final String COMP_NAME =
   "java:comp/env/ejb/SimpleSimulation";
 public static final String JNDI_NAME =
   "ejb/SimpleSimulation";

 public simulations.SimpleSimulation create()
                       throws javax.ejb.CreateException,
                              java.rmi.RemoteException;
}
```

XDoclet also generates the non-code artifacts (deployment descriptor in XML) and an aspect that is supplied to AspectJ. AspectJ performs the client modifications based on the generated aspect. Recall how the aspect code generated by the template of Figure 3-1 will change all object creation (`new SimpleSimulation()`) to calls to a remote object factory and all method calls (e.g. `sim.diffuse(plate);`) to calls to a remote interface.

Upon completion, GOTECH has generated a new EJB, deployed it in the application server, and modified the client code to interact with the new bean. The new distributed application can be used right away without requiring any additional configuration.

5.7 J-Orchestra Case Studies

To showcase J-Orchestra, we present four case studies of partitioning medium to large applications and of several smaller applications. The first three case studies demonstrate the benefits of appletizing by successfully transforming three realistic, third-party applications: JBits [27], JNotepad [36], and Jarminator [35], into client-server applications. JBits is not only the largest among all the case studies but also a commercial application available in bytecode-only form. While JNotepad and Jarminator are also third-party applications, they are free and publicly available. The last case study of partitioning the Kimura system [55] demonstrates how automatic partitioning can be an enabling technology for prototyping ubiquitous computing applications [97]. In this case study, the starting point is a distributed application that is rewritten with the explicit purpose to develop a centralized version that will later become distributed with J-Orchestra. Finally, we present some of the most representative smaller applications we have partitioned with J-Orchestra.

5.7.1 Appletizing Case Studies

In our measurements, we compare the partitioned applications' behavior to using a remote X display [71] 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.

5.7.1.1 JBits

JBits, the largest of the applications, is an FPGA simulator by Xilinx—a web search shows many instances of industrial use. The JBits GUI (see [27] 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, and so forth. 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.

5.7.1.2 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, run-

ning 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.

5.7.1.3 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 site, 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.

5.7.1.4 Discussion

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.

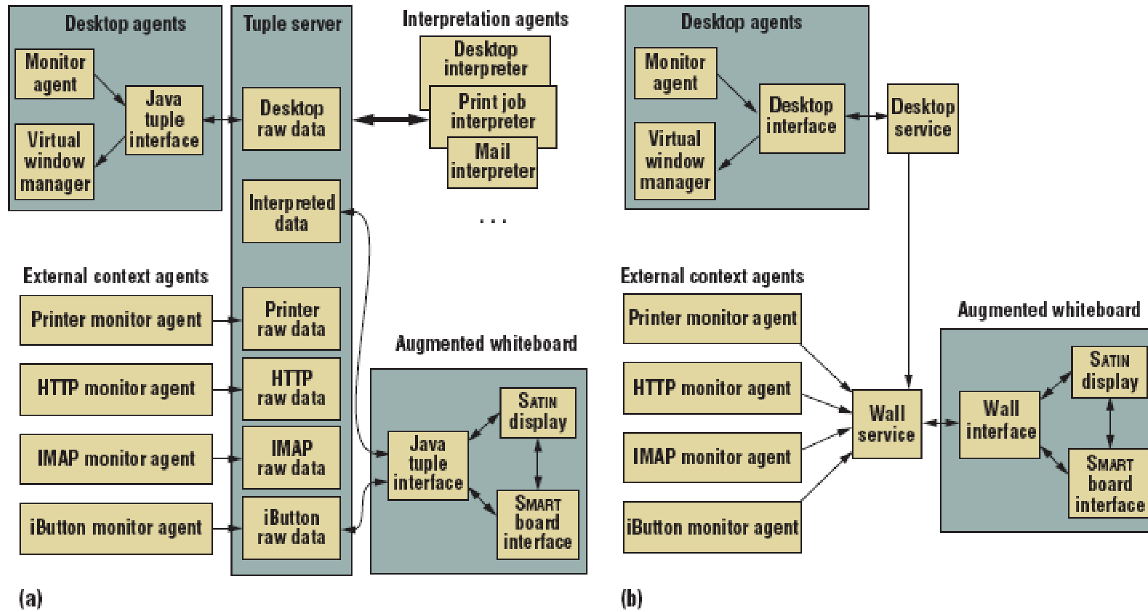


Figure 5-2: Kimura architecture: (a) the original system; (b) the reengineered Kimura2 system.

Our experience confirms the benefits of appletizing. Indeed, it 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.

5.7.2 Kimura Case Study

The Kimura case study [51] stands apart from other J-Orchestra case studies because its primary objective was not only to showcase the capabilities of J-Orchestra but also to explore whether automatic application partitioning can help researchers rapidly prototype distributed ubiquitous computing systems. Proponents of ubiquitous computing (or ubicomp, for short) [97] envision a future in which computers are inexpensive and plentiful

and seamlessly interoperate. Ubicom is also one area in which researchers have clearly identified the need for software engineering support [1]: although hardware continues to become smaller and cheaper, the corresponding software tools that would make the vision of ubicomp possible have not matured at the same rate. One major feature of the ubicomp domain—distinguishing it from traditional desktop applications, for example—is the software's inherent distributed nature. Ubicomp environments are naturally distributed over multiple computers connected via a wired or wireless network. These computers come in many shapes and sizes, from handheld to wall-sized. Applications in these environments are typically designed under the assumption that computing resources come and go in ever-changing combinations of lightweight and heavyweight, predefined and ad hoc groups. So, ubicomp application developers typically must suffer all the complexities of distributed-systems programming.

The difficulties that developers encounter when building ubicomp applications are more pronounced during research and prototype development. Ubicomp application prototypes are typically exploratory: The application's structure, the kind of data being shared, and the data's distribution characteristics will change frequently as the application undergoes iterations through the design-build-deploy-evaluate-redesign cycle. To facilitate application prototyping in this domain, developers must be able to modify the data structures' underlying distribution characteristics with little effort. Unfortunately, ubicomp developers often aren't expert at distributed systems. As a result, ubicomp researchers need simple, automated techniques that support rapid prototyping in such domains, and the Kimura case study explores how useful automated application partitioning with J-Orchestra can be in this respect.

As a larger case study of applying automatic partitioning to ubiquitous computing systems, we used automatic partitioning in developing the latest version of the Kimura system, which is a realistic, complex ubicomp application [55], Kimura is part of a research project that seeks to explore and evaluate the addition of visual peripheral displays to human-computer interfaces. Kimura uses large, projected displays as peripheral interfaces to complement an existing work area—the area surrounding a traditional desktop computer. Kimura uses these peripheral displays to help users manage multiple activities, such as coherent sets of tasks typically involving multiple documents, tools, or communications. Kimura assists in visualizing background activities as montages of images on the peripheral displays. These montages serve as anchors for background awareness information collected from a context-aware infrastructure.

Kimura's source code consists of 98 Java application classes and over 4,400 source statements. These application classes use many system and third-party classes, including Swing and Java Advanced Imaging (JAI) library classes, as well as classes that facilitate two-way communication with an electronic whiteboard.

The architecture of the original version of Kimura consists of three distinct components (see Figure 5-2a). A desktop interface module runs on the user's PC, monitoring all window and application activity through a native library and providing virtual-desktop functionality. A context interpreter module acts as an intermediate layer, aggregating the incoming messages from the desktop and the context-aware infrastructure and conveying them to the peripheral-display module, which we informally call "the wall." The wall, which connects directly to several projectors and a SMART Board interactive whiteboard,

maintains two-way communication with the SMART Board and provides up-to-date visualizations of the user's working contexts as montages projected onto the SMART Board.

These three components connect through TSpaces, a communication package designed to connect distinct distributed components [49]. TSpaces is based on the well-known tuplespace paradigm and incorporates database features such as transactions, persistent data, and flexible queries. It employs the publish-subscribe model. When one component adds or deletes a tuple on the TSpaces server, an appropriate callback method is called asynchronously in any other component that has registered to receive notifications matching that type of tuple.

The creators of TSpaces aimed at "hitting the distributed computing sweet spot [49]." The system lets programmers ignore many hard aspects of distributed communication, such as naming, state, and load balancing. The original Kimura implementation didn't use any of these advanced TSpaces features but employed TSpaces as a convenient way to keep shared state and to broadcast global events—such as activity changes—to all system components.

To evaluate the applicability of automatic partitioning to the ubicomp domain, we reengineered Kimura by removing the existing distribution code and redistributing it with automatic partitioning. The first step of reengineering was to separate Kimura's main application tasks from its network communication. In this way, we could create a simpler Kimura core, evolve it as necessary, and automatically partition it with J-Orchestra.

We first removed the code that supported distribution with TSpaces and replaced it with a single shared data structure. The result was a single program that could run in one process and open multiple windows—the wall and the desktop control panel—on a single

machine. The TSpaces-related code—functions responsible for connecting to the TSpaces server and adding or deleting tuples—was spread over 11 of the 77 source files. While TSpaces dictated an event-based structure for the application, the centralized version could use direct method calls between components, resulting in simpler, cleaner code.

Similarly, the interpreter component, which acted as an extra level of indirection between the desktop and the wall, was superfluous in the centralized version. We removed it as a distinct entity, preserving its functionality in two new modules designed to act as public interfaces of the desktop and the wall. These two new modules were two singleton classes whose responsibilities included handling incoming and outgoing messages from the application's other part. Coding and integrating them with the rest of the system was straightforward. As Figure 5-2b illustrates, Kimura's new version no longer has a central server. Instead, the system components talk to one another directly and synchronously.

Kimura2 consists of two partitions—one for the desktop and one for the peripheral display. The user interaction takes place through the peripheral display, while the desktop machine does the core of the processing, such as monitoring open applications. One can think of the peripheral display as a "monitoring console" for the Kimura working environment.

To make partitioning possible, we had to understand the application's internal structure, as the type based analysis heuristics of J-Orchestra, which determines what references can leak to what code, turned out to be too conservative in this case. Kimura2 uses Swing UI classes on what would become the wall and the desktop partitions. Because the code handling these objects is unmodifiable, we need to be sure that the objects in one partition are not shared in the other. Otherwise, the Swing code might try to access a remote object's

fields directly, resulting in a crash. The heuristic analysis conservatively concluded that Swing classes can't exist on two different partitions. However, we know that the Swing object partitioning in Kimura2 is safe: the Swing widgets on the desktop display are distinct from the Swing widgets on the wall display. Therefore, we could explicitly direct J-Orchestra to produce appropriate code for Swing classes on both partitions.

Altogether, of the 64 automatically rewritten classes, 43 were Swing and Abstract Window Toolkit (AWT) classes, and 6 were made serializable so that they could be by-copy passed by-copy across different memory spaces to improve performance. We excluded 71 Kimura application, 4 third-party, and 12 Java development kit (JDK) classes from the distribution process altogether because we determined that they never participate in the distributed communication. All in all, including testing, it took us only a few days to partition Kimura2 with J-Orchestra.

Discussion

Automatic partitioning turned out to be quite beneficial in the case of developing Kimura2. The main benefit is in the new software architecture's simplicity, which resulted in more understandable and maintainable code without sacrificing any of the original functionality. Kimura2's architecture will facilitate planned additions to the system much more easily because the developers can focus on the desired functionality without worrying about the distribution specifics. The new version also is easier to deploy because we don't need to maintain a running TSpaces server.

Table 5-7. Software Complexity Metrics

	Kimura	Kimura2	Percent more in original
Total statements	4,436	4,084	8.6
Number of classes	98	92	6.5
Number of methods	693	682	1.6
Program difficulty metric	3,305	3,124	5.8
Development effort metric	2,611	2,235	16.8
Lack of cohesion of methods metric	2,395	2,165	10.6
Interpackage fan-out (no. of classes)	881	822	7.2

To quantify this simplicity's benefits, we used JStyle [38] to derive software metrics. The software engineering community is still divided on software metrics' value and meaning, so the significance of our qualitative findings is somewhat subject to individual interpretation. Table 5-7 lists some of the more pronounced differences between Kimura and Kimura2. The new version exhibited better results in all metrics, including those not described in detail here.

Kimura originally consisted of 4,436 source statements (including declarations but not counting comments, empty statements, empty blocks, closing brackets, or method signatures). Out of them, 3,836 (86 percent of the total) remained unchanged in the new version. We completely removed the TSpaces-related code (486 statements, almost 11 percent of the total) and added 134 statements. Finally, we modified 114 statements to adapt the application to the new communication paradigm.

As Table 5-7 shows, the new version exhibited significant differences using the Halstead program difficulty metric [28], Chidamber and Kemerer's lack of cohesion of methods (LCOM) [16], and class fan-out (the number of classes a given class depends on). The new version is significantly less complex. Of course, we would expect a centralized architecture to be much less complex than a distributed one. However, it is interesting to quantify the difference.

In our evaluation of Kimura2, we also performed extensive measurements to evaluate how the partitioning infrastructure affects performance. Most system operations (including montage creation, montage switching, and document manipulation) exhibited significant speedup in relation to their counterparts in the original version, with only two of the measured operations (wall montage switching and document activation) showing a slowdown. We omitted our performance measurements because they are not essential to our conceptual evaluation of automatic partitioning for ubicomp, being merely the result of orthogonal, low-level concerns, such as the underlying middleware used in the case of J-Orchestra relative to TSpaces.

Our experiences using automatic partitioning to develop ubicomp applications have been quite positive. The approach's overwhelming advantages include both the simplicity of coding for a single machine without the need for distributed programming and the ease of repartitioning and redeployment. Furthermore, the ability to run on unmodified runtime systems—that is, any Java VM—is invaluable when using a multitude of heterogeneous devices. Nevertheless, we have also identified several shortcomings associated with automatic partitioning for ubicomp applications but not necessarily revealed by the Kimura case study. Most of these shortcomings stem from the fact that many general engineering

issues are difficult to address using an automated approach that J-Orchestra follows. Contrary to this automated approach, which involves no programming, just resource-location assignment (e.g., graphics code should run on this machine, or the main engine should run on that machine), a semiautomatic approach could let the user annotate detailed parts of the code and data that would actuate advanced distributed systems mechanisms (e.g., what data should be replicated, how the copies should remain consistent, and where leases should be used for fault tolerance). Indeed, a semiautomatic approach could resolve many of the issues associated with automatic partitioning, and we discuss this research direction in the future work section (Chapter VIII).

5.7.3 Other J-Orchestra Case Studies

Some of the most representative other applications we have partitioned to demonstrate J-Orchestra include:

- the Java Speech API demo mentioned in Section 4.2 on page 63. Speech is produced on one machine while the application GUI is running on a handheld (IPaq machine). In general, Java sound APIs can easily be separated from an application's logic using J-Orchestra.

- JShell: a third-party command shell for Java. The command parsing is done on one machine, while the commands are executed on another.

- PowerPoint controller: we have written a small Java GUI application that controls MS PowerPoint through its COM interface. We partitioned the GUI of this application from its back-end. We run the GUI on a IPaq PDA with a wireless card and use it to control a Windows laptop. We have given multiple presentations using this tool.

- A remote load monitoring application: machine load statistics are collected and filtered locally with all the results forwarded to a handheld (IPaq) machine over a wireless connection and displayed graphically. The original application was written to run on a single Windows machine.

This chapter discussed the issues of applicability of the three software tools for separating distribution concerns explored by this dissertation. We identified programming scenarios under which NRMI, GOTECH, and J-Orchestra would be most useful. Finally, we presented case studies that showcased how the tools can successfully separate distribution concerns of realistic applications.