Aspectizing Server-Side Distribution

Eli Tilevich Stephan Urbanski Yannis Smaragdakis
College of Computing, Georgia Institute of Technology
Atlanta, GA 30332
{tilevich, stephan, yannis}@cc.gatech.edu

Marc Fleury
The JBoss Group
Atlanta, GA
marc@jboss.org

Abstract

We discuss how a collection of domain-specific and domain-independent tools can be combined to "aspectize" the distributed character of server-side applications, to a much greater extent than with prior efforts. Specifically, we present a framework that can be used with a large class of unaware applications to turn their objects into distributed objects with minimal programming effort. Our framework is developed on top of three main components: AspectJ (a high-level aspect language), XDoclet (a low-level aspect language), and NRMI (a middleware facility that makes remote calls behave more like local calls). We discuss why each of the three components offers unique advantages and is necessary for an elegant solution, why our approach is general, and how it constitutes a significant improvement over past efforts to isolate distribution concerns.

1. Introduction

Separation of concerns [5] is the Holy Grail of computing. The term refers to the decomposition of a problem so that different facets are isolated from each other and reasoning can be performed independently. "Separation of concerns" has been a valuable philosophical mantra for educating computer scientists. Nevertheless, at the implementation level it has well-defined limits. There are concerns that fundamentally define what we mean by a computation, and, thus, cannot be separated. For instance, parallel algorithms often have no resemblance to sequential algorithms for the same problem and some problems are very unlikely to even have an efficient parallel solution. Thus "efficient parallelism" is not a concern that can be separated from the logic of a software application.

In view of such difficulties, most research has shifted from the problem of separating concerns to the problem of removing low-level technical barriers to the separation of concerns, assuming that the separation is conceptually possible. In language tools, two main directions have been identified. The first is that of general-purpose tools for expressing different concerns as distinct code entities and composing them together. The second is that of domain-specific tools that achieve separation of concerns for well-defined domains by hiding such concerns behind language constructs. The term "aspect-oriented" is often used to describe the first direction (although it was originally [8] proposed as a concept that encompasses both directions).

In this paper we present a general framework for separating distribution concerns from application logic. Our approach is a mixture of aspect-oriented techniques and domain-specific tools. Just like all other research in aspect-orientation, our goal is to remove low-level technical barriers to the separation of distribution concerns—the assumption remains that the structure of the application is amenable to adding distribution. The specific technical substrate that we target is that of server-side Java applications as captured by the J2EE specification. This domain is technically challenging (due to complex conventions) and has been particularly important for applied software development in the last decade. We show how a combination of three tools can yield very powerful separation of distribution concerns in a server-side application. We call this separation "aspectization", following other aspect-oriented work. (We use the main aspect-oriented programming terms in this paper, but do not embrace the full terminology. E.g. we avoid the AOP meaning of the term "component" as a complement of "aspect" [8].)

To classify our approach, we can distinguish between three levels of aspectization of a certain concern or feature:

- ***Type 1: "out-of-sight"***. The application already exhibits the desired feature. The challenge of aspectization is to remove the relevant code and encapsulate it in a different entity (*aspect*) that is composable with the rest of the code at will. The approach is application-specific.
- ***Type 2: "enabling"***. The application does not exhibit the desired feature, but its structure is largely amenable to the addition of the feature. Code implementing the feature needs to be added in a separate aspect but glue code may also need to be written to adapt the application logic and interfaces to the feature.

- ***Type 3: "reusable mechanism"***. Both the feature implementation and the glue code are packaged in a reusable entity that can be applied to multiple applications. Adapting an existing application to include the desired feature is trivial (e.g. a few annotations at the right places).

Our framework achieves Type 3 aspectization for a large class of server-side applications. In contrast, the closest prior work [14] attempts Type 1 aspectization and identifies several difficulties with the tools used: the need to write code to synchronize views, the need to create application-specific interfaces for redirecting calls, etc. These difficulties are resolved automatically with our approach. To achieve our goals we use three tools:

- ***NRMI*** [17]: an alternative to Java RMI that offers an efficient implementation of call-by-copy-restore semantics, in addition to regular call-by-copy. NRMI is the key for going from a Type 1 aspectization to a Type 2. That is, it provides the mechanism for enabling an application that is written without distribution in mind to be distributed without significant changes to its logic. The NRMI semantics is indistinguishable from local execution for a large class of applications—e.g. all applications with single-threaded clients and stateless servers.
- ***AspectJ*** [9]: a high-level aspect language. It is used as a back-end, i.e. our framework generates AspectJ code. It eliminates a lot of the complexity of writing glue code to turn regular Java objects into Enterprise Java Beans (*EJB*s).
- ***XDoclet*** [21]: a low-level aspect language. It is used primarily for generating the AspectJ glue code that adapts the application to the conventions of the distribution middleware. Like AspectJ, XDoclet is a widely available tool and our framework just provides XDoclet templates for our task. XDoclet is the key for going from a Type 2 aspectization to a Type 3. That is, it lets us capture the essence of the rewrite in a reusable template, applicable to multiple applications.

As an example, we used our framework, (called *GOTECH*, for "General Object To EJB Conversion Helper") to turn an existing scientific application (a thermal plate simulator) into a distributed application. The application-specific code required for the distribution consists of only a few lines of annotations. The rest of the distribution-specific code is provided by the GOTECH framework.

2. Background

We discuss next the challenges to adding distribution to an existing application and give the necessary background for our approach.

2.1. Challenges of Distribution

It is sometimes under debate whether distribution is a concern that can be at all separated from application logic. For example, Waldo et al.'s well-known paper [19] argues that "papering over the network" is ill-advised. The main reasons include difference in performance, different calling semantics, and the possibility of partial failure. (Other reasons mentioned in [19], like direct memory access, no longer hold today with languages like Java and C#.) Nevertheless, the real-world success of many projects and tools (e.g. the NFS distributed file system) is due exactly to the fact that they are "papering over the network", allowing use by unaware applications, even if the integration with distribution is not entirely seamless.

2.1.1. Semantics. Consider a centralized application written in a modular fashion with separate objects handling distinct parts of the application's functionality. 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. Such an emulation would be prohibitively slow, however. 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, etc.) on a different machine. 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, accessing each other's data (or OS-level resources, like I/O) directly and depending on reading the latest values of these data. In this case efficient distribution is impossible without change to the application structure.

Therefore, the assumption of our approach is that the application is amenable to adding distribution without fundamentally changing the application structure. In this case, the memory semantics issue can be alleviated by giving control to the programmer over the calling semantics and by emulating local semantics under certain assumptions (see Section 2.2.1.) 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, there is no general solution, 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.

2.1.2. 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 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, it is desirable to give to the programmer complete control over the location of objects.

2.1.3. Conventions. It has become a common business practice to use a middleware mechanism such as RMI, CORBA, DCOM, etc. to enable distribution. Since our work aims to remove the low-level technical barriers to aspectizing distribution, our main challenge is to change application code to interface with distribution middleware. This entails manipulating code to make it follow established conventions.

In object-oriented distributed systems, types are often used to mark an object to be able to interact with the middleware runtime services. For example, in order to be able 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 changing those methods of the object that are to be invoked remotely to declare that they could throw exceptions for network errors. The client code needs to be changed as well. A call to a remote object constructor might have to be replaced by a sequence of calls to a registry service. All of those changes can be quite tedious to apply. Tool vendors have made some inroads in alleviating the task of converting plain objects to 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 apply changes *to the clients* of the modified object.

2.2. The Elements of our Approach

2.2.1. NRMI. Most middleware (e.g., RMI, CORBA, etc.) offer *call-by-copy* semantics for remote calls. This means that when a reference parameter is passed as an argument to a remote routine, all data reachable from the reference are deep-copied to the server side. The server then operates on the copy. Any changes made to the deep copy of the argument-reachable data are not propagated back to the client, unless the user explicitly arranges to do so (e.g., by passing the data back as part of the return value).

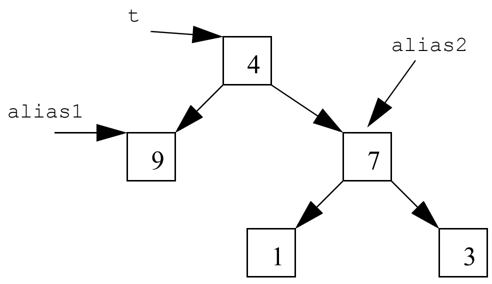

Figure 1. A tree data structure `t` and two aliasing references to its internal nodes.

NRMI (Natural Remote Method Invocation) [17] is a modified version of the Java RMI, such that the user can select *call-by-copy-restore* semantics for object types in remote calls, in addition to the standard call-by-copy semantics of RMI. Informally, call-by-copy-restore semantics means the following: First, the callee gets a copy of all data reachable by the caller-supplied arguments. Then, after the call, all modifications to the copied data are reproduced on the original data, overwriting the original data values in-place.

NRMI supports copy-restore semantics for arbitrary linked data structures (e.g. linked lists, trees, hash tables). The result is that remote calls behave much like local calls for most practical purposes. To use an example from [17], imagine the following function running on a remote server:

```
void foo(Tree tree) {
  tree.left.data = 0;
  tree.right.data = 9;
  tree.right.right.data = 8;
  tree.left = null;
  Tree temp = new Tree(2,tree.right.right, null);
  tree.right.right = null;
  tree.right = temp;
}
```

If the structure pointed by `t` in Figure 1 gets passed to this remote method `foo`, the result of the call on the client site (Figure 2) will be indistinguishable under NRMI and under local execution (i.e. if the client and `foo` were in the same address space). The reason is that NRMI will reproduce all remote changes to the local data in such a way that they will be visible even by other references (*aliases*) to the same data. NRMI does this while allowing the execution of method `foo` to proceed at full speed. Only after the end of the execution of the remote method, the new values of all objects reachable before the call will be

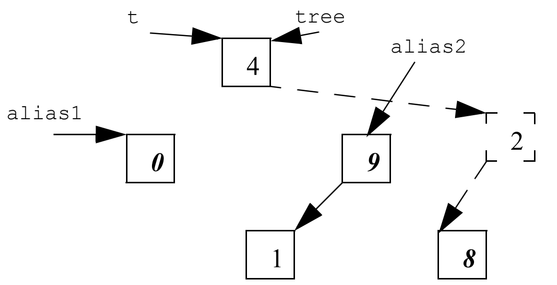

Figure 2. Changes introduced after the execution of foo, both for NRMI and for a local call. Even changes to now unreachable (from `t`) data are reproduced correctly.

examined and all changes will be reproduced on the client site. The algorithms and implementation of NRMI are discussed in detail in [17].

We should point out that NRMI is not a *Distributed Shared Memory* system, i.e. it does not try to emulate a single memory space across machines. Consequently, an NRMI remote call occasionally differs from a local call. Specifically, if the server code keeps references to the data passed as arguments after the end of the remote call, these data will not be kept consistent with the client's version. Furthermore, if the remote call is not atomic (i.e. if another thread in the client could access the argument data while a remote call takes place) the NRMI execution will be different from a local call. In this case, the programmer needs to be aware of the distribution.

Overall, however, NRMI succeeds in making remote calls resemble local calls for many practical scenarios. For example, in the common case of a single-threaded client (multiple clients may exist but not as threads in the same process) and a stateless or memory-less server, NRMI calls are indistinguishable from local calls.

The issue of reproducing the changes introduced by remote calls is important in aspectizing distribution. For instance, Soares et al. write in [14]:

When implementing the client-side aspect we had also to deal with the synchronization of object states. This was necessary because RMI supports only a copy parameter passing mechanism ...

and

[Reproducing remote changes] requires some tedious code to be written ...

With NRMI, the need for writing explicit code to reproduce remote changes is mostly eliminated. Thus, our approach can be more easily applied to unaware applications.

2.2.2. AspectJ. AspectJ [9] is a general purpose, high-level, aspect-oriented tool for Java. AspectJ allows the user to define aspects as code entities that can then be merged (*weaved*) with the rest of the application code. The power of AspectJ comes from the variety of changes it allows to existing Java code. With AspectJ, the user can add superclasses and interfaces to existing classes and can interpose arbitrary code to method executions, field references, exception throwing, and more. Complex enabling predicates can be used to determine whether code should be interposed at a certain point. Such predicates can include, for instance, information on the identity of the caller and callee, whether a call to a method is made while a call to a certain different method is on the stack, etc.

For a simple example of the syntax of AspectJ, consider the code below:

```
aspect CaptureUpdateCallsToA {
  static int num_updates = 0;

  pointcut updates(A a):
      target(a) && call(public * update*(..));

  after(A a): updates(a) {         // advice
    num_updates++; // update was just performed
  }
}
```

The above code defines an aspect that just counts the number of calls to methods whose name begins with "update" on objects of type `A`. The "pointcut" definition specifies where the aspect code will tie together with the main application code. The exact code ("advice") will execute after each call to an "update" method.

2.2.3. XDoclet. XDoclet is a widely used, open-source, extensible code generation engine [21]. XDoclet is often used to automatically generate wrapper code (especially EJB-related) given the source of a Java class. XDoclet works by parsing Java source files and meta-data (annotations inside Java comments) in the source code. Output is generated by using XDoclet template files that contain XML-style tags to access information from the source code. These tags effectively define a low-level aspect language. For instance, tags include `forAllClassesInPackage`, `forAllClassMethods`, `methodType`, etc. XDoclet comes with a collection of predefined templates for common tasks (e.g. EJB code generation). Writing new templates allows arbitrary processing of a Java file at the syntax level. Creating new annotations effectively extends the Java syntax in a limited way.

3. Our Framework

3.1. Overview

The GOTECH framework offers the programmer an annotation language[1] for describing which classes of the original application need to be converted into EJBs and how (e.g. where on the network they need to be placed and what distribution semantics they support). The EJBs are then generated and deployed in an *application server*: a run-time system taking care of caching, distribution, persistence, etc. of EJBs. The result is a server-side application following the J2EE specification—the predominant server-side standard.

The importance of using EJBs as our distribution substrate is dual. First, it is the most mature technology for server-side development, and as such it has practical interest. Second, it has a higher technical complexity than middleware such as RMI. Thus, we show that our approach is powerful enough to handle near-arbitrary technical complications—our aspectization task is significantly more complex than that of [14] in terms of low-level interfacing.

Converting an existing Java class to conform to the EJB protocol requires several changes and extensions. An EJB consists of the following parts:

- the actual bean class implementing the functionality
- a home interface to access life cycle methods (creation, termination, state transitions, persistent storing, etc.)
- a remote interface for the clients to access the bean
- a deployment descriptor (XML-formatted meta-data for application deployment).

In our approach this means deriving an EJB from the original class, generating the necessary interfaces and the deployment descriptor and finally redirecting all the calls to the original class from anywhere in the client to the newly created remote interface. The process of adding distribution consists of the following steps:

1. The programmer introduces annotations in the source
2. XDoclet processes the annotations and generates the aspect code for AspectJ
3. XDoclet does the EJB generation
4. XDoclet generates the EJB interface and deployment descriptor
5. AspectJ compiler compiles all generated code (including regular EJB code and AspectJ aspect code from step 1) to introduce distribution to the client by redirecting all client calls to the EJB instead of the original object.

(The XDoclet templates used in step 4 are among the pre-defined XDoclet templates and not part of the GOTECH framework.)

1. The annotations are introduced in Java source comments as "JavaDoc tags". We use the term "annotation" instead of the term "tag" as much as possible to prevent confusion with the XDoclet "tags", i.e. the XDoclet aspect-language keywords, like `forAllClass-Methods`.

3.2. Framework Specifics

We discuss many of the technical specifics of GOTECH in this section. Further examples can be found in Section 4.1., where we present an example application.

3.2.1. Middleware. In our development we used the JBoss open-source application server. JBoss is one of the most widely used application servers with 2 million downloads in 2002. Although our approach would work with other application servers, they would need to somehow integrate NRMI. (An alternative discussed in Section 3.3. is to have XDoclet insert the right NRMI code in the application. This just changes the packaging of the code but not the need for NRMI, and it is technically much more convoluted.) We have integrated NRMI in the JBoss code base as a middleware option and GOTECH uses it just like any other client would. We briefly describe this implementation for reference purposes, since it differs from the previously published implementation of NRMI [17].

Our original implementation of NRMI [17] was as a drop-in replacement for RMI. This required modification of the Java runtime system, which is undesirable for use in commercial software development. This prompted us to offer a complete, portable re-implementation of NRMI in JBoss. Fortunately, the JBoss server architecture is designed to be highly modular and extensible [13]. A remote invocation goes through a sequence of client and server interceptors. An interceptor performs some work with an Invocation object and passes it to the next interceptor in the chain. What interceptors are used for a particular remote method invocation is specified in the JBoss configuration type for each type of request/response protocol such as JRMP, IIOP, HTTP, or SOAP. NRMI was added simply by providing a client and a server interceptor and adding them to the chain of interceptors in the JBoss configuration file. Thus, NRMI was implemented in JBoss without having to modify any standard JDK classes and without having to understand the inner workings of the JBoss server.

3.2.2. GOTECH Annotations. In our approach, the programmer needs to provide annotations to guide the automated transformation process. Some of these annotations are EJB-specific (i.e. processed by existing XDoclet templates). Additionally, we added annotations for making remote calls use NRMI. Integrating copy-restore semantics required an extension of the JBoss-specific deployment descriptor. For instance, the following annotations will make a parameter passed using call-by-copy-restore. (This is a per-method annotation.)

```
public aspect GOTECH_<className/>WrapperAspect
  pertarget(target(<className/>)
       && (!cflow(within(GOTECH_<className/>WrapperAspect)))) {
   // Part I above: per-target aspect that captures object creation.
 private <classTagValue tagName="ejb:bean" paramName="interface-name"/> ep;

 GOTECH_<className/>WrapperAspect() {
  try {
   <classTagValue tagName="ejb:bean" paramName="name"/>Home sh;
   javax.naming.InitialContext initContext = new javax.naming.InitialContext();
   String JNDIName = "<classTagValue tagName="ejb:bean" paramName="jndi-name"/>";
   Object obj = initContext.lookup(JNDIName);
   sh = (<classTagValue tagName="ejb:bean" paramName="name"/>Home)
          javax.rmi.PortableRemoteObject.narrow( obj,
                              <classTagValue tagName="ejb:bean" paramName="name"/>Home.class);
   ep = sh.create();
  } catch (Exception e) { ... }
 }
   // Part II above: Intercepting object creation. A remote object factory is called. All access is through an interface.
 Object around() : target(<className/>)
                    && call(* *(..))
                    && (!cflow(within(GOTECH_<className/>WrapperAspect)))
 {
  try {
   Method meth = ep.getClass().getMethod(thisJoinPoint.getSignature().getName(),
                                    ((org.aspectj.lang.reflect.MethodSignature)
                                         thisJoinPoint.getSignature()).getParameterTypes());
   Object result = meth.invoke(ep, thisJoinPoint.getArgs());
   return result;
  } catch (Exception e) { ... }
 }
   // Part III above: Intercepting method calls.
}
```

Figure 3. Simplified fragment of XDoclet template to generate the aspect code. Template parameters are shown emphasized. Their value is set by XDoclet based on program text or on user annotations in the source file.

```
/**
 * @ejb:interface-method view-type="remote"
 * @jboss:method-parameters copy-restore="true"
 */
```

Note that without invoking GOTECH the comments remain completely transparent to the original application.

3.2.3. GOTECH XDoclet Templates. After the programmer supplies all the necessary information we can use XDoclet to generate files. The first task XDoclet is used for is creating the source code for the client aspect. The generated aspect's role is to redirect all method calls to the original objects to now be performed on the appropriate EJB. Additionally, the original object should only be referred to through an interface and its creation should be done by a distributed object factory instead of through the operator `new`. (We ignore direct field reference for now, but it could be handled similarly using AspectJ constructs.) A simplified (shorter XML tags, eluded low-level details) fragment of our XDoclet template appears in Figure 3. The template file consists of plain text, in this case a basic AspectJ source file structure, and the XDoclet annotation parameters, whose value is determined by running XDoclet.

For ease of reference we have split the template in Figure 3 in three parts. Part I defines that the aspect is per-target, i.e. that a unique instance of the aspect will be created every time a target object (i.e. an instance of class `className`, which is derived from the `name` parameter we saw earlier) is created. The other conditions in Part I determine that the interception of the construction of a target object should only occur if this takes place outside the control flow of the Aspect itself. Note that the template uses XDoclet's ability to access class information (`<className/>`) in addition to user-supplied annotations.

Part II of the template shows the code that will be executed for the creation of a new instance of the aspect. This is the code that takes care of the remote creation of the EJB using a remote object factory mechanism.

Finally, Part III makes the generated aspect code capture all method calls (`call(* *(..))`) to objects of class `className` unless the calls come from within the Aspect itself.

The next task for XDoclet is to transform the existing class into a class conforming to the EJB protocol. To do this, we need to make the class extend the `SessionBean` interface. Additionally, all parameters of methods of an EJB must implement interface `Serializable`: a Java marker interface used to designate that the parameter's state can be "pickled" and transported to a remote site. We do this by creating an aspect that when run through AspectJ will make the parameter types implement interface `Serializable`. The template file for this transformation is not shown, but the functionality is not too complex.

The last task where we employ XDoclet is the generation of the home and remote interface as well as the deployment descriptors. XDoclet has predefined templates for this purpose. The only extension has to do with the copy-restore semantics and generating the right deployment descriptor to use NRMI. Note that this step needs to iterate over all methods of a class and replicate them in a generated interface, while adding a `throws RemoteException` clause to every method signature. This is a task that Soares et al. [14] had to perform manually in their effort to aspectize distribution with AspectJ. A simplified fragment of the XDoclet template for iterating over the methods appears below:

```
<forAllMethods>
 <ifIsInterfaceMethod interface="remote">
    public <methodType/> <methodName>
                            (<parameterList/> )
           <exceptionList append=
                   "java.rmi.RemoteException"/>;
 <ifIsInterfaceMethod>
</forAllMethods>
```

3.3. Discussion of Design

Our approach uses a combination of AspectJ, NRMI and XDoclet in order to add distribution to existing applications. Each tool has unique advantages and greatly simplifies our task. Of course, in terms of engineering choices, there are alternative approaches:

- instead of our three tools, we could have a single, special-purpose tool, like D [11], JavaParty [12] or AdJava [6] that will rewrite existing Java code and introduce new code and meta-data. (None of these tools deals with the EJB technology, but they are representatives of domain-specific tools for distribution.) We strongly prefer the GOTECH approach over such a "closed" software generator approach. The first reason is the use of widely available tools (AspectJ, XDoclet) that allow exposing the logic of the rewrite in terms of templates. Templates are significantly easier to understand and maintain than the source code of a compiler-level tool. The second advantage of our approach is the use of inobtrusive annotations inside Java source comments. The original Java program can be used just like before when no addition of distribution functionality is required.
- we could have XDoclet generate all the code, completely replacing both NRMI and AspectJ. In the case of NRMI, this would mean that XDoclet will act as an inliner/specializer: the NRMI logic would be added to the program code, perhaps specialized as appropriate for the specific remote call. Conceptually, this is not a different approach (the copy-restore semantics is preserved) but in engineering terms it would add a lot of complexity to XDoclet templates. Similar arguments apply to the use of XDoclet to replace AspectJ. AspectJ allows manipulations taking Java semantics into account—e.g. the `cflow` construct mostly used for recognizing calls under the control flow of another call (i.e. while the latter is still on the execution stack). Although the emulation of this construct with a run-time flag is not too complex conceptually, it does require essentially replicating the functionality of AspectJ in a low-level, inconvenient, and hard-to-maintain way. XDoclet is not designed for such complex program manipulations.

Finally, one could ask whether a combination of AspectJ and NRMI without XDoclet would be sufficient. Unfortunately, this approach would suffer a more severe form of the drawbacks identified by Soares et al. [14]. These drawbacks include needing to write the remote interface code by hand, not being able to work without availability of source code, etc. The problem is exacerbated in our case because our target platform (EJBs) is more complex and because we are attempting complete automation. To automate the construction of EJBs, we need to generate the remote and home interfaces from the original class, as well as generate non-code artifacts (the deployment descriptor meta-data in XML form). None of these activities could be automatically handled by AspectJ. In general, low-level generation, like iterating over all methods and replicating them (with minor changes) in a new class or interface, is impossible with AspectJ. The same is true for "destructive" changes, like adding a `throws` clause to existing methods.

4. Applying the Framework

4.1. Example Application

In this section we present an example of applying the GOTECH framework to convert a scientific application into a distributed application interacting with an applica-

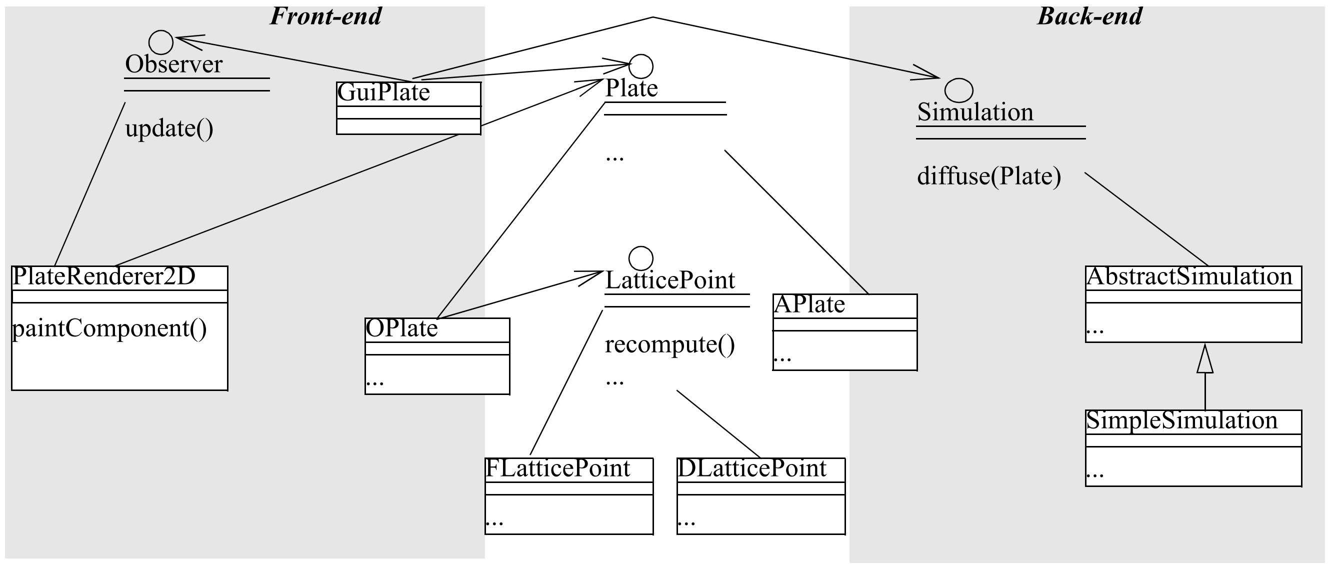

Figure 4. UML class diagram of the Thermal Plate Simulator functionality

tion 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. [14]. (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 4. 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` 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 `SimpleSimulation` 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 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.

4.2. Advantages and Limitations

4.2.1. Advantages of our approach. Despite the simplicity of applying GOTECH, the resulting code is feature-by-feature analogous to that written manually by Soares et al. [14]. We discuss each element of the implementation and perform a comparison.

Making the object remote. With GOTECH, this step is quite simple. A new remote interface is created from the original class using XDoclet. Soares et al. identified several problems when trying to perform the same task with AspectJ, even though their original application already supported reference to the relevant objects through an interface. Specifically, Soares et al. could not add a `RemoteException` declaration to the constructor of their "facade" class (which is analogous to our `SimpleSimulation` class) using AspectJ. In our approach, the original class does not need to be modified: a slightly altered copy forms the bean part of the EJB. It is easy to add exception declarations when the new class gets created (see the `exceptionList append` statement in Section 3.2.).

Serializing types. Soares et al. needed to write by hand (listing all affected classes!) the aspect code that will make application classes extend the `java.io.Serializable` interface so they can be used as parameters of a remote method. In their paper, they acknowledge:

This might indeed be repetitive and tedious, suggesting that either AspectJ should have more powerful metaprogramming constructs or code analysis and generation tools would be helpful for better supporting this development step.

Indeed, our approach fulfills this need. Using XDoclet, we create automatically the aspect code to make the parameter types implement `java.io.Serializable`.

Client call redirection. The code introduced by the generated aspect of Figure 3 (part III) does a similar redirection as with the technique of Soares et al. That is, it executes a call to the same method, with the same arguments, but with a different target (a remote interface instead of the original local reference). Nevertheless, in the Soares et al. technique this code had to be introduced manually for each individual method. These authors admit:

... [T]his solution works well but we lose generality and have to write much more tedious code. It is also not good with respect to software maintenance: for every new facade method, we should write an associated advice....

We should note that it is not really XDoclet or NRMI that give us this advantage over the Soares et al. approach. Instead, our aspect code of Figure 3 (part III) uses Java reflection to overcome the type incompatibilities arising with a direct call. This technique is also applicable to the Soares et al. approach.

Updating Remotely Changed Data. NRMI offers a very general way to update local data after a remote method changes them. Our approach is not only more general than the one used by Soares et al. but also more efficient. Specifically, Soares et al. admit the need to "synchronize object states". They perform this task by trapping every call to an update method, storing the affected objects in a data structure, and eventually iterating over this data structure on the remote site and reproducing all the introduced changes. NRMI is a more general version of this technique, applicable to a large class of applications. The Health Watcher system of Soares et al. is one of them: the system is "non-concurrent" (as characterized by the authors) and the two sites do not need to always maintain consistent copies of data: it is enough to reproduce changes introduced by a remote call. Soares et al. acknowledge both the need for automation and the fact that the structure of state synchronization in Health Watcher is general:

... it would be helpful to have a code analysis and generation tool that would help the programmer in implementing this aspect for different systems complying to the same architecture of the Health Watcher system.

Additionally, NRMI is more efficient than capturing all calls to update methods. Instead of intercepting every update call, NRMI allows the remote call to proceed at full speed and only after the end of its execution it collects the changed data. (To do this, before execution of the remote call, NRMI needs to store pointers to all data reachable by parameters. This is not costly, since these data are transferred over the network anyway.) Soares et al. admit the inefficiency of their approach, although they argue it does not matter for the case of Health Watcher.

4.2.2. Limitations. Currently the GOTECH framework suffers from some engineering limitations. We outline them below. Some of these limitations are shared by the approach of Soares et al.—assuming that this approach is applied to multiple applications. Recall, however, that our templates only automate some tedious tasks. Although these templates are not application-specific, they also do not attempt complete coverage for all Java language features. In general, it is up to the programmer to ensure that the GOTECH process is applicable to the application.

Entity Bean support. So far we have only concentrated on distributing the computation of an application. Thus, we only have templates for generating Session Beans and not Entity Beans. Entity Beans are commonly used for representing database data through an object view. There is no further technical difficulty in producing templates for Entity Beans, but their value is questionable in our case. First, we are not aware of an example where adding distribution to an existing application requires creating any Entity Beans. Second, the Entity Bean generation will have more constraints than Session Beans—for instance, Entity Beans should support identity operations (retrieval by primary key) since they are meant for use with databases. These operations usually cannot be supplied automatically—the original class will have to support such operations, or a fairly complex XDoclet annotation could supply the needed information.

Conditions for applying rewrite. Our aspect code controlling where we apply indirection in the original code is currently coarse grained. Consider again Part I of Figure 3. The generated aspect code is applied everywhere except in points in the execution under the control flow of the EJB. This roughly means that our approach assumes that the desired distributed application is split into a client site and a server site, and the server site never calls back to the client. On the server site, the calls to the existing class are not redirected. The positive side-effect of this rule is that server-side objects communicate with each other directly, thus suffering no overhead. Future versions could have a finer grained control over when the indirection should be applicable.

Exceptions, construction, field access. There are some more minor engineering issues with the current state of our templates. For instance, the handling of remote method exceptions is generic and cannot be influenced by the programmer at this stage. This is just a matter of regular Java programming: we need to let user code register exception handlers which will get called from the `catch` clauses of our generated code. Another shortcoming of our template of Figure 3 is that it only supports zero-argument constructors. (This is fine for stateless Session Beans, which by convention have no-argument constructors.) There is a simple rewrite that can alleviate the problem and we plan to use it. We also currently have no support for adding indirection to direct field access from the client object to the remote object. This should be quite feasible with AspectJ, and we intend to add this capability. Nevertheless, direct access to fields of another object may mean that the two objects are tightly coupled, suggesting that perhaps they should not be split in the distributed version.

Note that all of these issues are relatively easy to fix. Since GOTECH templates are easy to inspect and change, application programmers can even incorporate this functionality on a per-application basis.

Finally, since performance is an important concern, we should emphasize that it is not an issue for the GOTECH framework. For the most part, GOTECH just generates the code that a programmer would otherwise add by hand. Additionally, in the only case where something is done automatically (when using NRMI) the mechanism is quite optimized [17]. In general, however, for a given set of distribution and caching decisions, the constant computational overheads of a distribution mechanism like ours are relatively unimportant. These overheads are small relative to the inherent cost of communication (including network time and middleware, e.g., EJB, overheads). These costs are not important if only few objects are accessed remotely. On the other hand, if many objects are accessed remotely, any distribution mechanism will suffer.

5. Related Work

We will separate related work into directly related work and indirectly related work. Directly related work includes other tools that help the programmer in adding distribution, but without taking away control and responsibility of the distribution process from the programmer. Such tools are "distributed programming aids": they help do the tedious tasks that the programmer would otherwise need to do manually and that would "pollute" the code describing the application logic. Nevertheless, the programmer is still responsible for ensuring that the tools do the right job for the application at hand.

Indirectly related work includes mostly application partitioning tools and Distributed Shared Memory systems. Such tools offer a higher-level interface. Their user does not necessarily program the distributed application, but rather offers hints to improve its performance. These tools have a higher correctness responsibility: they attempt to correctly distribute *any* application although they usually result in loss of efficiency and are applicable in fewer situations than the "distributed programming aids".

5.1. Comparison with Directly Related Work

Many domain-specific languages have been proposed to aid distributed programming, and some of them [8][11] were key examples in the early steps of Aspect Oriented Programming. Each of the domain-specific languages for distribution offers its own advantages. For example, D [11] and Doorastha [4] concentrate on allowing fine-grained control over how data get passed to remote sites: both by-copy and by-reference semantics can be selected and classes can describe which of their fields get passed remotely. JavaParty [12] is a higher-level tool than D and Doorastha, offering generality and enabling object mobility, but without giving the programmer much control over the distribution process. The FarGo [7] system is similar to JavaParty but with less emphasis on generality with respect to language features and more emphasis on object movement as a group.

Compared to such domain-specific tools for distribution, the GOTECH framework offers several advantages:

- it is an easy to evolve tool, based on widely used aspect-oriented infrastructure (AspectJ and XDoclet). Inspecting and changing the functionality of our XDoclet templates is much easier than changing the code for any of the above domain-specific tools.
- it offers NRMI, which is a unique way to support a remote call semantics that is closer to local execution. NRMI is applicable to many common scenarios and eliminates the need for explicitly updating data when changes are introduced by remote calls.
- GOTECH targets EJBs as a distribution substrate. This is a more complex, industrial-strength technology than the middleware used by the above systems.

5.2. Indirectly Related Work

There are many tools that attempt to offer distribution capabilities to existing programs. Such tools include Distributed Shared Memory (DSM) systems and application partitioning tools. DSM systems [1][2][3][20] offer an abstraction of shared memory to applications running over a network of machines. In order to obtain acceptable performance, DSMs employ user annotations and relaxed semantics: rigorously defined assumptions about the way the application updates shared data that if violated will result in incorrect execution.

Partitioning systems (like Pangaea [15], our own J-Orchestra [16], and Addistant [18]) are similar to DSMs but emphasize offering a shared memory abstraction through rewrites of the application. The benefit is that the re-written application can be executed on an unmodified runtime system (e.g. a regular Java VM).

Both DSMs and partitioning systems operate at a much higher level than tools like GOTECH. They strive for correct distributed execution of all applications and give the programmer much less control over distribution choices. Therefore, just like very high level languages, these tools are valuable for the cases where they are applicable, but these cases are a small part of the general distributed computing landscape. In contrast, GOTECH does not take away control from the programmer, but instead offers assistance for generating tedious code that would otherwise be intertwined with the application logic.

Finally, other researchers have examined the suitability of aspect-oriented techniques for different domains. For example, Kienzle and Guerraoui [10] examined the suitability of aspect-oriented tools for separating transaction logic from application logic. Separating transaction processing from application logic is very hard, and possible only under very strict assumptions about the application. These findings of Kienzle and Guerraoui are consistent with longtime observations of the database community.

6. Future Work and Conclusions

We presented the GOTECH framework: an approach to aspectizing distribution concerns. GOTECH relieves the programmer from performing many of the tedious tasks associated with distribution. GOTECH relies on NRMI: a middleware implementation that makes remote calls behave much like local calls for a large class of uses (e.g. single-threaded access to client data and no memory of past call arguments on the server). Additionally, GOTECH only depends on general-purpose tools and offers an easy to evolve implementation, easily amenable to inspection and change. Compared with the closest past approaches, GOTECH is significantly more convenient and general.

In high-level terms, GOTECH is also interesting as an instance of a collaboration of generative and aspect-oriented techniques. The generative elements of GOTECH are very simple exactly because AspectJ handles much of the complexity of where to apply transformations and how. On the other hand, AspectJ alone would not suffice to implement GOTECH.

There are several directions of future work, both in improving the framework and in providing more mature support for the conversion of plain objects to EJBs with different tools. For instance, part of the upcoming work on the JBoss application server includes bytecode engineering at class load time to retrofit existing classes so that they become EJBs. This approach can be applied both to distribution and to persistence concerns and is of high industrial value. Since NRMI is already part of JBoss, this bytecode engineering work can result in a replication of the GOTECH capabilities at load-time. Another promising direction for more mature use of GOTECH includes developing analysis tools that formalize the preconditions for the applicability of the approach and ensure they are met by a specific application.

Availability and Acknowledgments

All source code for the GOTECH framework and the thermal plate simulator application is publicly available.

This work was supported by the NSF (under Grants 0220248 and 0238289) and by the Yamacraw Foundation.

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

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

[5] Edsger W. Dijkstra, "On the role of scientific thought", EWD 447, August 1974. Also in *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.

[6] Mohammad M. Fuad and Michael J. Oudshoorn, "AdJava— Automatic Distribution of Java Applications", 25th *Australasian Computer Science Conference (ACSC)*, 2002.

[7] Ophir Holder, Israel Ben-Shaul, and Hovav Gazit, "Dynamic Layout of Distributed Applications in FarGo", *Int. Conf. on Softw. Engineering (ICSE)* 1999.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, "Aspect-Oriented Programming", *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[9] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, "An Overview of AspectJ", *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[10] Joerg Kienzle and Rachid Guerraoui, "AOP: Does It Make Sense? The Case of Concurrency and Failures", *European Conference on Object-Oriented Programming (ECOOP)*, Malaga, June 2002.

[11] Cristina Videira Lopes and Gregor Kiczales, "D: A Language Framework for Distributed Programming", PARC Technical report, February 97, SPL97-010 P9710047.

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

[13] Francisco Reverbel and Marc Fleury, "The JBoss Extensible Server", *ACM Middleware 2003*.

[14] Sergio Soares, Eduardo Laureano, Paulo Borba, "Implementing Distribution and Persistence Aspects with AspectJ", *OOPSLA* 2002.

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

[16] Eli Tilevich and Yannis Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning", *European Conference on Object-Oriented Programming (ECOOP)*, Malaga, June 2002.

[17] Eli Tilevich and Yannis Smaragdakis, "NRMI: Natural and Efficient Middleware", *Int. Conf. on Distributed Computer Systems (ICDCS)*, 2003. Extended version available from `http://www.cc.gatech.edu/~yannis`.

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

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

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

[21] `www.xdoclet.org`