CHAPTER III

GOTECH

This chapter describes GOTECH, a framework that can be used with a large class of unaware applications to turn their objects into distributed objects with minimal programming effort. GOTECH combines domain-specific and domain-independent tools to "aspectize" the distributed character of server-side applications to a much greater extent than with prior efforts. Specifically, the GOTECH framework has been 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 the GOTECH approach is general, and how it constitutes a significant improvement over past efforts to isolate distribution concerns.

3.1 Introduction

GOTECH (for "**G**eneral **O**bject **T**o **E**JB **C**onversion **H**elper") is a general framework for separating distribution concerns from application logic in enterprise Java applications via a mixture of aspect-oriented techniques and domain-specific tools. Following the objective of removing low-level technical barriers to the separation of distribution concerns, the framework operates under the assumption that the structure of the application is amenable to adding distribution. GOTECH targets the specific technical substrate of server-side Java applications as captured by the J2EE specification [78]. This domain is technically challenging (due to complex conventions) and has been particularly important for applied software development in the last decade.

This work demonstrates 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 chapter, but do not embrace the full terminology. E.g. we avoid the AOP meaning of the term “component” as a complement of “aspect” [40].)

To classify the GOTECH 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).

The GOTECH framework achieves Type 3 aspectization for a large class of server-side applications. In contrast, the closest prior work [72] 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, and some others. GOTECH resolves these difficulties automatically. To achieve its goals, it uses three tools:

- ***NRMI*** [88]: a middleware mechanism described in detail in the previous chapter. 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*** [41]: 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) [78].
- ***XDoclet*** [103]: 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 (see Chapter V for a detailed description), we used GOTECH 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 GOTECH framework provides the rest of the distribution-specific code.

3.2 The Elements of Our Approach

3.2.1 NRMI

The issue of reproducing the changes introduced by remote calls is important in aspectizing distribution. For instance, Soares et al. write in [72]:

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 ...

Our NRMI middleware, described in detail in Chapter II, 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. 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.

3.2.2 AspectJ

AspectJ [41] 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, and so forth.

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.

3.2.3 XDoclet

XDoclet is a widely used, open-source, extensible code generation engine [103]. 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`, and so forth.

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.

3.3 The Framework

3.3.1 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 [78] 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, and so forth of EJBs. The result is a server-side application following the J2EE specification [78]—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 [72] 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:

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 `forAllClassMethods`.

- the actual bean class implementing the functionality
- a home interface to access life cycle methods (creation, termination, state transitions, persistent storing, and so forth)
- 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 generates the EJB
4. XDoclet generates the EJB interface and deployment descriptor
5. The 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.)

3.3.2 Framework Specifics

We discuss many of the technical specifics of GOTECH in this section. Further examples can be found in Chapter V, in which we present an example application.

3.3.2.1 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 3.3.3 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.) Section 2.5.2 of this dissertation describes in detail the integration of NRMI in the JBoss code base as a middleware option. GOTECH uses NRMI just like any other client would.

3.3.2.2 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.)

```
/**
 * @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.

3.3.2.3 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, elided low-level details) fragment of our XDoclet template appears in Figure 3-1. 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 3-1 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` XDoclet parameter) 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.

```
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 3-1: 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.

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 implement 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 in which 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. [72] 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>
```

3.3.3 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 [52], JavaParty [66] or AdJava [23] 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 unobtrusive annotations inside Java source comments. That is, the annotations do not affect the source code of the original Java program, which can still be used in its centralized form.
- 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. Similarly, one can imagine replacing all uses of AspectJ with more complex XDoclet templates. Yet AspectJ allows manipulations taking Java semantics into account—e.g. the `cflow` construct mostly used for recogniz-

ing 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. [72]. These drawbacks include needing to write the remote interface code by hand, not being able to work without availability of source code, and so forth. 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.

3.4 Advantages and Limitations

3.4.1 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. [72]. 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 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 3.3.2).

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 3-1 (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 3-1 (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.

3.4.2 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.

3.4.2.1 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.

3.4.2.2 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 3-1 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.

3.4.2.3 Making types serializable

Our current approach of making classes implement `java.io.Serializable` so that they could be passed as parameters to remote method calls works only for some application classes. Indeed, our current XDoclet template for generating aspects that adds `java.io.Serializable` to all non-serializable parameter types makes several assumptions.

One assumption is that having a type implement this marker interface is sufficient for making it serializable by Java Serialization [79]. However, this is not always the case: a type is serializable only if all the types reachable transitively from it are also serializable. Our current implementation performs no such check. Nevertheless, this is a reasonable assumption for a framework that assumes that the original centralized application is amenable for distribution in the first place. Making a type adhere to the serializability requirements could be non-trivial, requiring significant changes to its implementation. In that case, careful manual code restructuring often is the only feasible option for performing these changes.

Another assumption is that all non-serializable parameters are application classes. In other words, all system JDK classes, passed as parameters to a remote method, must be serializable, for applying aspects to system classes has not been standardized. Even if changing the implementation of a system class (i.e., having it implement an additional interface) were straightforward, that would violate the design principles of our framework, creating the need for a custom runtime environment. However, in practice it never makes sense to modify the serializability properties of a system class. Because significant effort has gone into designing system classes that are part of the standard JDK, the ones that are serializable are always marked as such (i.e., implementing `java.io.Serializable`).[2] The system classes that are not serializable are usually the ones that control some local system resources such as threads, sound, etc. It would be meaningless to send instances of such classes over the network anyway. Thus, making a centralized program amenable to our approach requires restructuring it in such a way that no non-serializable system classes are used as parameters in remote methods.

2. In addition, instances of some serializable system classes could become invalid if serialized and transferred to a machine on a different network node. E.g., `java.io.File`.

3.4.2.4 Exceptions, construction, field access

The current state of our templates leaves some more minor engineering issues unresolved. 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 that will get called from the `catch` clauses of our generated code. Another shortcoming of our template of Figure 3-1 is that it only supports zero-argument constructors. (This is fine for stateless Session Beans, which by convention have no-argument constructors.) However, it would be only a matter of engineering to implement an additional rewrite to address this problem. We also currently have no support for adding indirection to direct field access from the client object to the remote object, which should be quite feasible with AspectJ. 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.

The subset of implemented functionality in the current version of GOTECH is sufficient to illustrate our approach. At the same time, all of the remaining issues would be relatively easy to address in a production system—GOTECH templates are highly amenable to inspection and modification. In fact, it is quite feasible that application programmers would incorporate additional functionality to GOTECH 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 in which something is done automatically (when using NRMI) the mechanism is quite optimized [88]. 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.

3.5 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.

Section 5.6 presents an example of applying the GOTECH framework to convert a centralized scientific application into a distributed application interacting with an application server.