Turning Java Components into CORBA™ Components with Replication

Stu Barrett, Phillip Foster

One of CORBA's™(Object Management Group, Inc.) key selling points over RMI and DCOM is that it can be used to "wrap" legacy applications that run on a wide range of platforms and are written in a wide range of languages.

This paper reviews some of the problems encountered while using CORBA™ to port an existing Java application. The application chosen was a simple Document Repository application, written as a test case for this effort. We took the approach of first writing the application in a "single process" fashion, debugging, and then porting the application to a distributed CORBA™. This had the advantage that we could focus on the application logic in a localized environment without the distractions and complications of a distributed environment.

Another goal of this effort, driven by a primary goal of the Object Infrastructure Project, was to use this distributed application as a test bed for the investigation of how we could seamlessly add non-functional "'illities" to the application's components. To that end, after the application was ported to CORBA™, we seamlessly added a replication mechanism so that distributed replicas of a Repository would share in the updates made to any of their peer replicas. This replication service was the realization of the "high availability" non-functional requirement for the Repository.

Port to CORBA™:

After we built and tested a working set of "single process" Java code, we were very interested in how to use IDL/CORBA™ to build the distributed wrappers to allow those Java objects to be accessed by a remote client.

We had goals that in hindsight were naïve:

- Keep existing implementation code intact as much as possible. Since the implementation code was already working, there should not be a need to modify it. Furthermore, since the original implementation was done with the expectation of distribution this should be an easy task.

- Hide portions of the implementation that we did not want the users to be aware of by simply not exposing those characteristics (e.g. attributes and methods) in IDL.

IDL:

The IDL file that specified the interfaces of the Java components was generated using a utility that read the Java files. This file was then hand crafted to remove translation errors and to prune any “private” aspects of the interfaces.

We should review the output of the IDL2Java compiler since this really frames the problem/solution space. First look at a sample piece of IDL that was created to “publish” the set of operations that we wanted to support.

The application’s IDL is shown in Figure 1. Consider, in greater detail, the IDL for method AddDocument in interface Repository

```
Module DocRepos {

        …………

interface Repository {

        …………..

 Document AddDocument( in string DocumentName,
                       in string Version,
                       in User Owner,
                       in string Description,
                       in DocumentStream userDocStream)
        raises( DuplicateDocument, StreamError );

        …………..
};
```

This IDL illustrates some important mechanisms that need to be understood in order to port the Document Repository application:

- Interfaces passed as arguments.
- Interfaces returned by operations.
- Exceptions.

The output of the IDL compiler resulted in the creation of a plethora of Java files (for ***each*** interface). These files defined the following types of information.

- Java interface defines the method signatures for the interface.
- Stub Class used by client.
- Skeleton Class used by implementation.
- TIE class used for delegation to implementation.
- Helper Class contains static methods that provide utility methods.
- Holder Class is “Container class” used to handle “out” and “inout” parameters.

Java.Util IDL issues:

An early realization was that some of the Java implementation methods returned objects of type Vector. This is a natural way in Java to handle lists. While we could have continued with this approach (define the IDL for the Vector interface), we felt that doing so would result in extremely poor performance due to superfluous distributed method calls. We modified the implementation code to return an array and then defined the IDL return as an IDL sequence.

This gave us an example of the common problem of distributing an existing application: in a single process environment, references are cheap, but in a distributed system, we have to be careful about sending a reference when we really want the data. OMG is looking at how to extend CORBA™ to support "pass by value" in addition to "pass by reference" (which, by the way, RMI already supports).

TIE wrapper approach:

There are two mechanisms that the implementation can use to "plug-in" to CORBA™ object adapter services. One way is to define your implementation classes so that extend the IDL generated "skeleton" classes. When you do that then your implementation objects *are* CORBA™ objects. The other was is to use separate CORBA™/TIE objects..

The TIE object extends from the CORBA™ skeleton object and is therefor a CORBA™ object. It delegates all method calls to an associated implementation object. All object references that are returned to the client need to be wrapped with a TIE object.

We felt that using the TIE approach would help us keep our implementation objects "pure".

Also of consideration was that one of the approaches under investigation for adding "'illities" to components is to implement appropriate services within the CORBA Stub and/or Skeleton wrappers. We felt that on the implementation side, the TIE wrapper objects would provide a good "platform" for the implementation of those services.

We used our own TIE wrapper (OIP_TIE) objects since we wanted to be able to put in our replication hooks.

One important concept to understand is that the implementation code had its own object definitions and the methods on those objects manipulated objects within that "object space". Many objects contained attributes that held object references to other objects with the Document Repository "object space". Most of the implementation methods had object references as part of their parameter list signatures or as return values. The implementation (as is the wont of typical OO design techniques) manipulated references to other implementation objects .

This means that you can think of the CORBA™ server as having two types of objects: Implementation objects, and TIE objects that are created, on demand, to provide access to a subset of the implementation objects at any one time. For example, if the implementation object space is populated from a persistent store (at startup time), the only TIE object in existence is the “root” object (i.e. The Repository object). As clients navigate through the implementation object space and object references are passed back, OIF_TIE object wrappers are dynamically created.   This dichotomy approach will also help with scaling issues (see Fine Grain Object Issue)

Porting Issues:

Signature problems:

The first problem we ran into was the fact that the ***methods*** for the classes of the existing code (implementation classes) have different signatures than the ***interfaces*** that are generated by the IDL compiler (IDL classes/interfaces). One of the differences that was apparent at first was the IDL interfaces were scoped in a different package than the implementation objects.

This is not a problem unless your object methods return object references or have object references as parameters. Unfortunately most of the Document Repository IDL operations are of this type.

The TIE wrapper objects implement the IDL interface. The ***constructor*** for the Repository TIE wrapper requires that the delegate object be of type DocRepos.RepositoryOperations. This means that the implementation objects ***must*** have method signatures that correspond to the IDL (e.g. all object references must be of type DocRepos.XXX). While this may now appear to be obvious, it exposed our goal of non-modification of the implementation code to be naïve.

Lets look at the signature changes needed. First look at the unmodified (and abridged!) implementation code:

```
public class Repository {

         …..
 public Document AddDocument(  String DocumentName,
                               String Version,
                               User Owner,
                               String Description,
                                        DocumentStream userDocStream)
      throws DuplicateDocument, StreamError {

      …………
}}
```

Now look at how the code was modified with respect to signatures.

```
public class Repository implements DocRepos.RepositoryOperations{
      ………
public DocRepos.Document AddDocument( String DocumentName,
                                     String Version,
                                     DocRepos.User Owner,
                                     String Description,
                                     DocRepos.DocumentStream userDocStream)
      throws DocRepos.DuplicateDocument, DocRepos.StreamError
{

      …………
}}
```

Note that the class definition was modified to indicate that the class implements the DocRepos.RepositoryOperations interface. Also note the AddDocument method signature was modified to match the method declaration of that interface.

TIE wrapper approach:
All implementation objects that needed to be made visible to clients needed to have an OIP_TIE object created. This includes objects that are created via a "factory" method (eg. AddDocument) as well as all of the objects references that are returned in a sequence (eg. ListDocuments). This means that the implementation methods that return object references to implementation objects need to wrap that object(s) with a TIE wrapper before it is returned.

Example:

```
      return newDoc;
```

Needs to be:

```
      return new OIP_TIE_Document(newDoc);
```

Fine Grain object issue:

While it would have been possible for us to hide the TIE object using a smart constructor for the implementation objects which would create a TIE object as a side effect of creating the implementation object, we would have had an explosion of TIE objects; one for each fine grain implementation object, regardless of whether that object was referenced by a remote client. By deferring the creation of the TIE object until remote object reference was needed we allow for a more scaleable server implementation. This is an example of tradeoff between transparency of implementation code and performance.

Access to private implementation functionality:

A problem arises with object references that are passed as parameters. The IDL only defines a subset of the functionality of the implementation objects. For example, implementation objects contain attributes and methods that are needed for proper implementation, yet they are not defined in IDL.

This presents a problem with IDL object references that are passed as parameters and the implementation code that needs to access the actual implementation object to gain access to a non-IDL piece of functionality. The object reference is to an object that supports the IDL interface, not the actual implementation interface.

In order to do get around this problem, we needed to access implementation class via the _delegate variable in the TIE object. Luckily this is only needed when the implementation code needs to access the implementation object directly and bypass the TIE wrapper.

Call Back mechanism:

Another major change was needed in order to implement the “call back” mechanism. In order for files to actually be moved from client to server, the client needed to provide a callback object (DocumentStream) so that the server can read the contents of the file on the clients machine. This required the client to create the object (with TIE wrapper), and register it with the BOA. The client then needed to spawn off a thread for that object to be serviced by the BOA event loop. While this is straight forward, it is another example of how the code changes when clients are remote.

Miscellaneous:

We had to make changes in the implementation code that had methods that defined “out” or “inout” parameters since the IDL compiler represented these parameters with “holder” classes. This is not necessary in the “single process” version of the application since these types of parameters are passed by reference.

We had to change the method signatures that threw exceptions so that they matched the generated IDL exception classes. We had to make changes so that all of the exception objects that were thrown or caught were the IDL class. Exceptions were defined in the IDL module.

Adding Replication:

The approach that we used to implement the replication service was loosely based on the Anti-Entropy mechanisms that were used on the Bayou project. [1]

This approach puts the responsibility of replication on the server not the client. When the server processes a request from a client that causes its implementation object space to mutate, it will “share” that request with its replicas so that they can maintain synchronization. The implementation of the replication mechanism required the use of persistent event channels, talker and listener objects/threads and the definition of the application specific anti-entropy protocols.

The relevancy to the components topic is that our approach was done in a manner that was transparent to the implementation and client programs. We were able to add this replication feature to an existing set of components with only minor modifications, none of which required changes to the components implementation.

Since we were using OIP_TIE wrappers to delegate each request/method call to the implementation object, we were positioned to simply make extensions to that delegation wrapper. That extension composed an anti-entropy protocol message using the method arguments and broadcasted the message to all of the replicas using the event channel.

Conclusions and Summary:

Hindsight shows us that the goal of porting an existing application code base to CORBA™ w/o making changes to the code base was naive. While it may have been possible to solve this problem with another set of wrappers, that effort would have been at least as much work as making changes to the implementation code.

A better way (although not possible with “legacy” code) would have been to define the IDL first, then write the implementation code.

By keeping CORBA™ Objects (i.e. TIE objects) separate from implementation objects we were able to de-couple their respective lifecycles and hopefully provide for a server implementation that was more scaleable.

Using the OIP_TIE wrappers, it ***was*** possible to add the replication feature to the implementation components after they were constructed. The OIP_TIE wrappers provided the “platform” for implementing the replication mechanism that was transparent to the client as well as the implementation code.

As with all coding efforts, the devil is in the details.

Acknowledgments:

Deborah Cobb: for implementing the persistent event channels used with the anti-entropy protocol.

References:

---

[1] Karin Petersen , Mike J. Spreitzer , Douglas B. Terry , Marvin M. Theimer and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. *SOSP '97***.** (C) ACM 1997

http://www.parc.xerox.com/csl/projects/bayou/pubs/sosp-97/