

Abstract

We present the generic facilities in $\hat{\Omega}$ for providing compile-free access to methods of CORBA servers and implementing CORBA servers in the $\hat{\Omega}$ language. These are dynamic in nature using the CORBA Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI), respectively. Other CORBA tools available in $\hat{\Omega}$ are described. Approaches to generalizing these mechanisms to other interpreted languages such as *S*, *R* and *Xlisp-Stat* are also (briefly) discussed.

This assumes a very basic knowledge of Client/Server architecture and some CORBA terms. I will explain them briefly as I go.

1 Running the Example

In order to set the scene, here are the instructions to run the example.

- Change directory to `$OMEGA_HOME/Interfaces/CORBA/DynamicJavaObject/Docs/`.
- Start the Orbacus naming service.

```
nsserv -OApport 5001 &
```

- Start the Interface Repository (IR) and trap the IOR to the file `/tmp/IR.ref`.

```
irserv -i -OApport 500 > /tmp/IR.ref &
```

- Populate the IR with the IDL module defined in `TraceServer.idl`.

```
make idlfeed
```

which expands to

```
irfeed -ORBrepository `cat /tmp/IR.ref` TraceServer.idl
```

You can check the contents of the repository using the $\hat{\Omega}$ [IRViewer](#) tool

```
omega -CORBA -ix IRViewer
```

- Run the server object process.

```
omega -CORBA -i server
```

- Run a CORBA-enhanced $\hat{\Omega}$ evaluator. For the moment, change the `evaluatorClass` property to [org.omegahat.Interfaces.CORBA.Pa](#) and invoke $\hat{\Omega}$ as

```
omega -CORBA
```

- Finally, issue the following Omegahat commands

```
localClasses().add(new URL("http://www.omegahat.org/Jama.zip"));
attach(1); // make the naming service a local database.
source("MatrixFunctions.omg"); // provide the Omega functions
Server.simple();
Server.foo("Any old string whose length will be returned");
Server.trace(Jama.Matrix.random(10, 10));
```

2 The Problem

Consider the following problem. We have a server object running on some machine and we want to utilize its services by invoking methods on it. The server is a CORBA object and the client must therefore communicate using the same CORBA mechanism. Our client is the interactive language $\hat{\Omega}$ ¹. This is an interactive and interpreted version of the Java language with several extensions. It provides direct interface to all Java classes and objects. The fact that it is an interpreted language implies that there are no direct CORBA/IDL compilers for it. Additionally, even if there were, we would not want to rely on them. We want to dynamically locate the server and discover and invoke its available methods without knowing them in advance and compiling stubs. The CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface provide a mechanism for doing this in $\hat{\Omega}$ and other similar languages such as *S*, *R*, *Xlisp-Stat* and *Matlab*. So the goal is no compilation of CORBA/IDL stubs or skeletons and easy implementation of CORBA clients and servers in an interactive language.

The particular problem we will use as our example is very stilted. However, it hopefully is simple enough to not obscure the CORBA details.

3 Server

The server has publically available methods that can be invoked by any client that can locate it. These methods are defined by the IDL interface(s) that the server implements. Our example declares 3 methods displayed below. (These are part of an IDL module defined in `TraceServer.idl`²).

```
interface TraceServer {
    double simple();
    long foo(in string val);
    double trace(in Matrix x);
};
```

The first two methods are basically useless and are just used as a test. They take 0 and 1 arguments respectively and involve only CORBA primitives and for the most part are not challenging to our generic CORBA interface from $\hat{\Omega}$. (These are handled by the basic [CorbaConverter](#).)

The `trace()` method is the one of interest. It takes a single argument, which is an IDL matrix, described by another CORBA/IDL interface:

```
interface Matrix {
    long nrow();
    long ncol();
    double data(in long i, in long j);
};
```

This exposes just enough of a numeric matrix to do most computations, adequate for computing its trace. (We will add another method for efficiency reasons below and illustrate different approaches.)

The implementation of the `trace()` method must be coded in terms of these three methods available from its argument. Up until now, the implementation is language neutral. A simple Java implementation of this is

```
public double trace(Matrix m) {
    int n = Math.min(m.nrow(), m.ncol());

    double total = 0.0;
    for(int i = 0 ; i < n ; i++ ) {
        total += m.data(i, i);
    }

    return(total);
}
```

¹See URL <http://www.omegahat.org>.

²See URL `TraceServer.idl`.

Note that there are three lines which make calls back to the argument. If this is in another process, these are CORBA method invocations. In this way, the client process has become a server process and specifically the Matrix argument is a server object.³ It is not necessarily true that the Matrix argument `m` is resident in the client process. It could come from a 3rd process or event from the same process as the server. Note also that these methods communicate primitive values (arguments and return values) and so we do not need to consider nested CORBA servers any further than 2 levels (the initial server and client). However, such nesting poses no conceptual problem in this setup.

4 Client

The client must first obtain a reference to the server. There are several ways to do this. The simplest is that the server has been registered with a CORBA naming service and that the client can locate it using some agreed name.⁴ The $\hat{\Omega}$ environment provides utilities to access the elements of the CORBA naming service as local variables. In our example, we will assume the server is identified by the name `Server` and that we can refer to it locally using this variable name. The actual mechanisms that we employ here can be quite different. The simplest approach is to use the specialized evaluator `org.omegahat.Interfaces.CORBA.Parser.OrbacusEvaluator` and the `attach()` to provide the top-level name server as a regular $\hat{\Omega}$ database. We will assume this is the case as it also provides a simple syntactic facility for invoking CORBA methods. A simple way to do this is to specify this evaluator class as the `evaluatorClassName` property in the `OmegaOptions` and to invoke the $\hat{\Omega}$ environment as

```
omega -CORBA
```

which provides the appropriate run-time arguments and configuration files.

First we will invoke the simple method `foo()` which we saw above. It takes an IDL string and, unknownst to us at this point, returns the length of the string. Because of our special CORBA-aware evaluator configuration, we can invoke this method with the command

```
Server.foo("An arbitrary string")
```

This will return the value 19, the length of the supplied string.

So let us examine the the underlying computations with the $\hat{\Omega}$ environment that executed this CORBA call. Firstly, because of the initial `attach()` call and the derived `Evaluator` currently in effect that is CORBA-aware, a `NamingServiceDatabase` for the top-level CORBA `NamingContext` has been attached to the evaluator's search path. The expression `Server.` identifies the variable named "`Server`" which is located by searching through the elements of the search path until such a name is found. This is found in the `NamingServiceDatabase` and so a CORBA `Object` is returned. The next component in the expression is identified as a method call, with the name `foo` and arguments `m`. When this is evaluated, there is no local Java method named `foo()` in the CORBA object. Because we are using a CORBA-aware parser, the method call expression takes care of looking for a suitable CORBA method named `foo()`. In this case, it finds one. It then creates a `CorbaCall` with the value of `Server` as the argument and `foo` as the name and "`An arbitrary string`" as the single argument. It must convert this argument to a CORBA object. Since this is a primitive, life is good and the default converter (`CorbaConverter`) handles this and constructs the appropriate argument list. Then the call is invoked and the return value awaited. Then it is converted from a CORBA object and into an $\hat{\Omega}$ object bu the same converter and returned.

And these are the basic steps for any CORBA method invocation.

- Identify target object and method definition.
- Convert the arguments to the types specified by the IDL operation definition.
- Invoke the method with the conversion arguments.
- Convert the return value to a local object.

³I

⁴Other methods allow a generic reference - an Interoperable Object Reference (IOR) - to be used.

4.1 Non-primitive Arguments

So at this point, we are ready for the non-primitive arguments and the `trace()` method. First, let us create a matrix object in $\hat{\Omega}$. For the purpose of exposition, we use the `Jama Matrix` class⁵.

```
localClasses().add("/home/duncan/Java/Jama")
m = Jama.Matrix.random()
```

So now we have an object with suitable data to be provided to the `trace()` method of `Server`. If we were to issue the command

```
Server.trace(m)
```

we would go through the same steps as the `cORBA` invocation above.

So what happens when convert the argument `m`. In the case of `String`, this was simple - the Java class had a direct correspondence with a CORBA class. In this case, the `StatMatrix.Matrix` class has no direct corresponding CORBA class. However, we can assume that the user is calling the method correctly (yeah, right!) and so we should convert the object to the type expected by the CORBA method. We can discover this from the Interface repository. This is the CORBA server which is used to house the IDL interfaces and modules that define the servers and arguments. The default converter used in this setup (based on the evaluator) creates a proxy object which declares that implements the required CORBA type - `StatMatrix::Matrix`. This is a dynamic object rather than a compiled version. It promises to attempt to honor requests for the IDL interface `StatMatrix::Matrix`.

And this contract or promise is the magic behind the dynamic CORBA implementation. It leaves us free to implement the request in an arbitrary fashion. We have two methods available to use. Firstly, we can attempt to evaluate a method invocation by looking for an corresponding function in the evaluator. An second alternative is to look for a method in the proxied object - the `Jama.Matrix` object `m` in this case. In our example, we choose to implement the requests by looking for `Function` objects with the same name as the CORBA operation name. So, for `nrow()` method, we look for a function named `nrow()` and we call it with the single argument `m`. Then we convert the return value to a `long`. Likewise, we look for `Functions` named `ncol()` and `data()`. In order to do this, we define functions for the three methods provided by the `Matrix`. These are specialized to expect `Jama.Matrix` as their sole argument. Then these compute the corresponding value.

```
function nrow(x) {
  x.getRowDimension();
}
```

Now, as requests are received, the `invoke()` method of the proxy object will evaluate the corresponding function. So we are simply providing a connection between the CORBA method name and an $\hat{\Omega}$ function that implements the functionality when given the local object (embedded in the proxy) and the remote CORBA arguments.

4.2 Implementation in S/R

For a language such as `S` (or `R`), the call might look something like

```
.Corba(server, "trace", matrix(random(100),10, 10))
```

In this case, the functions `nrow()` and `ncol()` are already available. We just have to create a function or method for `data()` which we do as

```
data = function(m,i, j) {
  m[i,j]
}
```

Note that in these languages there is no easy way to generically hide the `.Corba()` call. We might provide a wrapper method

⁵See URL .

```
setMethod("trace", signature("CORBAObject", "matrix")) {
  function(x, m)
    .Corba(x, "trace", m)
}
```

Now we can issue the call

```
trace(server,)
```

With some effort, we can implement the Name service as a database in the same way as is done in $\hat{\Omega}$ and so reference *Server()* directly. Otherwise, we must obtain the reference to the *Server* and hold it as a *CORBAObject* which is a stringified version of the object using its IOR.

4.3 Extended Approaches

One of the problems with the above approach is that we have defined functions in the $\hat{\Omega}$ name-space for *nrow()*, *ncol()* and *data()* that are implemented in terms of *Jama.Matrix*. This is basically polluting the $\hat{\Omega}$ namespace. There are a variety of ways around this which we discuss here.

One approach is to put these functions in their own *Database* object which is not an element of the search path. This is supplied (via the *CorbaConverter*) to the *BasicDynamiOmegaObject* proxy and its *invoke()* method arranges to temporarily attach this database or look for functions here when evaluating the request. There are problems with this approach.

When we attach the database, to which position do we *attach()* this database - 0 or 1? If we attach it in position 0, assignments will be committed to it rather than the intended or expected evaluation frame. While we can easily circumvent this using the facilities in the $\hat{\Omega}$ evaluators, it is not a good design. If we attach the database to position 1 or higher, it is possible that we will resolve the function to an unintended one of the same name.

We can implement the functionality by not attaching the database but looking within it to resolve the initial function object to satisfy the CORBA request. Doing this implies that the regular evaluation of that function will not be able to relocate itself or the other functions contained in the database. Hence recursive calls and calls to the other methods will fail or give incorrect results.

4.4 Generic Functions

So another approach is to define multiple versions of these functions that are to be called with different argument types. In other words, we overload the functions and allow the $\hat{\Omega}$ evaluation model locate the appropriate one for a particular call. We would define the functions for a *Jama.Matrix* as follows:

```
method ncol(Jama.Matrix m) {
  m.getColumnDimension();
}

method nrow(Jama.Matrix m) {
  m.getColumnDimension();
}

method data(Jama.Matrix m, i, j) {
  m.get(i, j);
}
```

Other definitions for any of these functions can be defined for the $\hat{\Omega}$ evaluator and will be stored with these and resolved during a particular call. (This is partially implemented in $\hat{\Omega}$ at present).

Note that we had to provide some type-information to ensure that the system could discriminate between the different functions. In the *data()*, we provided type information for the first method and allowed the other two to be of any class. Unfortunately, in this model, other functions can be added which would conflict with this and lead to ambiguity. This object model doesn't help the provider of methods in these cases and is most useful when there is no possibility of

ambiguity. It is basically syntactic sugar allowing like methods to be named the same and to have the system determine which one is appropriate. (The $\hat{\Omega}$ implementation differs from the S one in that the arguments will not be coerced to the appropriate type but will be passed unaltered by reference to the selected method. There is no general mechanism for coercion, but instead there is a “compatibility” concept implemented via the `isAssignableFrom()` in the `JavaTMClass` class.)

A potentially better solution for both the user and name-space pollution is to somehow bind the functions to the embedded Matrix object. The concept of a closure is useful here. We collect the functions into a database We assign the embedded object to a specific name in a new database and then we define the functions in that database and treat the embedded object as a free or global variable. The `nrow()` might be defined as

```
function nrow() {
    matrix.getColumnDimension();
}
```

Of course, the functions need to agree on a name for the embedded object. We could use the name `this` and then we have a basic object system. Grouping the functions together, along with one or more variables is closure.

This approach leads us naturally to consider implementing the CORBA methods in a Java class and using objects of this class as the embedded object in the proxy.

4.5 Java Method Implementations

One of the benefits of the approaches above is that code to implement a particular CORBA method is interpreted and can be modified dynamically and the $\hat{\Omega}$ debuggers can be employed to examine their behavior. While this is a good thing, it does suffer from speed issues. Additionally, it is unnecessary if the Java object acting as the server already contains methods with an appropriate signature which can be invoked by in response to a CORBA request (in the `invoke()` method of the proxy object). Well, in the future, we will be able to compile $\hat{\Omega}$ functions into Java methods and we will use this to compile an `OmegaUserClass` (i.e. interpreted) class. Before that functionality is implemented and when the Java class is already available, we would like to use it directly.

The `DynamicJavaObject` class does just this. It is very similar to the `BasicDynamicOmegaObject`. It takes a real `JavaTM` object for which it is to act as a proxy and the name of one or more IDL interface definitions. The difference between this class and `BasicDynamicOmegaObject` lies solely in its implementation of the `invoke()` method to handle CORBA requests. Rather than look for an $\hat{\Omega}$ `Function` with the same name as the CORBA operation being requested, it looks for a `JavaTM` method in the `JavaTM` object. Effectively, it replaces a call of the form

```
operationName(embeddedObject, ...)
```

with

```
embeddedObject.operationName(...)
```

It does this by examining the list of potential overloaded methods and finding the most appropriate one, allowing for the conversion of the arguments from CORBA objects to `JavaTM` objects. The basic method dispatching of the $\hat{\Omega}$ system is used for this.

We can construct a simple example of this. We define a new `JavaTM` class named `IDLMatrix`. This extends the JAMA matrix and defines the three methods

```
public class Matrix extends Jama.Matrix {
    // same constructors as above
}
```

Note that this is different from compiling the CORBA/IDL skeletons and implementing the CORBA-connectivity that way.

5 The Server Implementation

In the example, we implemented the server by generating and compiling the IDL skeletons and extending the appropriate class with methods that implemented the IDL methods. At this point, we have enough information to do it dynamically, either using a simpler Java class (non CORBA-aware) and avoiding the skeletons, or using the $\hat{\Omega}$ interpreter and functions.

To create the local Java class for TraceServer, we implement the three IDL methods. Of course, we need to compile the skeletons for the Matrix argument so that we can compile the Java class unless we use the dynamic CorbaCall classes in the body.

The interpreted version can be implemented by first defining three functions

```
function foo(server, arg) {
    return(arg.length());
}

function simple(server) {
    return(Math.random());
}

function trace(server, m) {
    local n = Math.min(nrow(m), m.ncol(m));
    int n;
    for(i = 0; < n ; i++) {
        total += data(m,i,i);
    }
}
```

Now, we initialize the ORB and BOA and attach the naming service.

```
attach(0);
```

At this point, everything is setup to evaluate each of the potential CORBA requests for a TraceServer object. In that interface, the server object needs no internal data of its own. So, we can actually pass a null object to the dynamic CORBA proxy and just tell it to implement the particular IDL interface - StatMatrix::TraceServer. We do this with the assignment to the variable `Server`. Since the naming server is attached in the default database position, the assignment is made there and hence the right-hand-side is visible on the CORBA bus via the binding "Server". Finally, we activate the server on the CORBA bus and allow it receive requests. There are two ways to do this. The `impl_is_ready()` will block until all server objects have been deactivated. The `init_servers()` is an Orbacus-specific facility which is a non-blocking call. In our world, we can continue to use the evaluator interactively and requests will also be received and processed. There is an issue here of synchronization which can be partially controlled via properties in the Orbacus tools controlling how requests and servers are threaded. See Orbacus Manual⁶ for more details

```
Server = new BasicDynamicOmegaObject(null, "IDL:StatMatrix/TraceServer:1.0");
boa.init_servers(); // or boa.impl_is_ready(null);
```

6 Drawbacks

The intent of the dynamic mechanism is to allow newly discovered CORBA objects to be employed without the need to locate the IDL code and compile the stubs and skeletons. This is very useful for prototyping the different parts of a system. Modifying the IDL in the Interface Repository does not necessarily require recompiling both server and client code.

⁶See URL <ftp://ftp.ooc.com/pub/OB/3.1/OB-3.1.2.pdf.gz>.

In many cases, however, the IDL is relatively static and extensions are made to it by inheritance (in the IDL). In these cases, we could compile the stubs and skeletons from the IDL and use CORBA in the more traditional ways. Then we are responsible for implementing the server by extending the skeleton class and providing the method bodies.

What does this effort buy us? The simple answer is run-time speed. The (*JavaTM*) Orbacus stub and skeleton implementations use the DSI and DII mechanisms. However, they utilize the compile time information of a method to populate the CORBA request object used in the dynamic methods.

The dynamic methods discussed here also populate such an object. However, they must also create the request object for each call. Basically, the cost of obtaining this object is an additional communication with the target server object or Interface Repository or both. (This is implemented via the

```
opDef = server._get_inerface_def(op_name);
orb().create_operation_list(opDef);
```

calls.) While we can cache these to improve performance, this is the efficiency we are giving up.

One aspect of the dynamic mechanism that we lose in the compiled version is the automated conversion of the method call's arguments. Of course this can be built into the compile the stubs and skeletons compiled version, but is not as immediate (especially given *JavaTM*'s lack of multiple inheritance).

So, as usual, there is no silver bullet and different situations will have different requirements - flexibility offered by dynamic implementations and improved speed provided by compiled stubs and skeletons.

7 Additional IDL Methods

The simple definition of the IDL *Matrix* interface provided sufficient functionality to implement most operations on a matrix, but not necessarily efficiently. The resulting implementation of the `trace()` method involved multiple CORBA invocations of the `data()` method. Assuming we started with an matrix argument of dimension n , the body of the `trace()` method involved $n + 2$ CORBA invocations. Implemented as dynamic CORBA servers using Ω functions, this may be expensive.

If we extend the definition of *Matrix* to return the entire contents of the matrix, we reduce the number of CORBA invocations. However, this is where more controversy is possible. How do we represent the contents? As 1 or 2 dimensional array? Arranged row or column-wise? There is no general satisfactory answer. We could provide a method for all styles...

For our purposes, we will return a 2 dimensional array organized row-wise, defined in IDL terms as

```
typedef sequence<double> DoubleSequence;
typedef sequence<DoubleSequence> TwoDDoubleArray;
TwoDDoubleArray values();
```

Now the `trace()` method can be implemented in *JavaTM* as

```
public double trace(Matrix m) {
    double[][] values = m.values();
    int n = Math.min(values.length, values[0].length);
    for(int i = 0 ; i < n ; i++) {
        total += values[i][i];
    }
    return(total);
}
```

The code looks very similar to the original implementation but involves just a single CORBA call, independent of the dimensions of the matrix. The tradeoff is one communication of a large amount of data versus multiple communications with less data in each. Different configurations will perform differently for the two setups. Compression and buffering come to mind as important factors.

We also have to account for the resources involved in computing the 2 dimensional array of values. The server may have to make a copy of the data and iterate over every value or if it is already stored in this fashion provide a simple reference.

8 Automated Generation of IDL