

This describes the basic overview of how CORBA services can be integrated into an interactive interpreted language such as *S* or *R*. It does not describe much about CORBA and assumes that the basics are known. Instead, it describes the implementation and the structure of the code that provides the low level connection to these languages. The hope is to reuse much of the code here across the two systems.

One aspect that the two environments (*S* and *R*) share is that the user-level language and variables are separated from their physical representation in memory, controlled as *C* structures. This is different from the $\hat{\Omega}$ environment and means that we must do a little more work here, especially in terms of robustness, etc.

The ideas are quite simple, especially given the working version in $\hat{\Omega}$. First, we establish the basic mechanism which allows the language invoke CORBA methods in a CORBA server. This presumably is residing in a different process and language at this point. We use the $\hat{\Omega}$ facilities to create such objects simply. Initially, we only deal with primitive type arguments and return values. Dealing with CORBA objects are return values is a trivial extension. Using 3rd party CORBA objects as arguments is also easy. Converting user-level objects to CORBA objects is the final step, and when done generically is very simple. Having completed this last step, we basically have the ability to make user-level objects CORBA servers.

What we do not do at this point is generate IDL from user-level methods and objects. While this is possible, it can be done by people who are familiar with the environment's language and have some familiarity with IDL. It is done in $\hat{\Omega}$ via the than in user-level objects in *S* and *RJava2IDL* than in user-level objects in *S* and *Rclasses*. There, there is a great deal more structure in the objects than in user-level objects in *S* and *R*. This is a consequence of the languages, not a deficiency.

1 Other CORBA Interfaces

Fnorb, CorbaScript, COPE are some examples of CORBA interfaces to languages other than *JavaTM* and *C++*. $\hat{\Omega}$ is, of course, in my opinion one of the more complete and useful interfaces. It does not require, but easily admits, skeletons and stubs. It treats CORBA objects as first class citizens, as local variables. It allows implementation of CORBA servers directly in the language, provides direct access to CORBA operations in remote servers and converts local objects to CORBA objects automatically. In addition to this, it provides complete access to Java objects, and a scripting language. It hides most of the CORBA details visible in these other tools and allows easy access to CORBA in an intuitive manner. These other tools do the same with a different focus and outlet (i.e. Python, Perl and ?) Most $\hat{\Omega}$ and the interface described here require no additional compilation, linking or wrappers.

2 Generality and Implementation

Much of the work was done a year ago using Visigenic's CORBA implementation. Some of it was Visigenic specific. I have used the Orbacus¹ implementation since then and it has worked relatively smoothly. The TAO² ORB may be better for a *C*-based implementation. For now, we use Orbacus. Let me know when things do not work with other implementations.

3 Basic CORBA Initialization

Before the environment can access any CORBA related services, it must initialize the connection to the ORB - the Object Request Broker. For many of the implementations, it must also initialize BOA - the Basic Object Adapter.

The initializers take an array of strings, usually the command line arguments of an application, and "extract" the relevant values. This allows things such as the location of the naming service, interface repository, etc. to be configured at run time.

The routines that can be called from the user-level to initialize the ORB and BOA are

```
void CORBAinit(char **args, long *length, long *status, long *statusLength)
s_object *S4_CORBAinit(s_object *args, s_object *indicators)
```

¹See URL <http://www.ooc.com/ob>.

²See URL <http://www.cs.wustl.edu/~schmidt>.

for $S3/R$ and S respectively³.

While the structure of the arguments are different, they have the same intention and effect. The first two arguments in the R version represent the command line arguments as a *character()* object. For $S3/R$, we must pass in the contents and then the length as two arguments. The other two arguments are logical/booleans (*logical()*) and should be of length 2. The values indicate whether to initialize the ORB and the BOA respectively. The routine reads the values and determines whether to initialize the associated component and then sets the value to indicate whether the operations were successful. In this way, it is an inout argument carrying information to the routine and a return status from the routine.

I believe we cannot convert the ORB or BOA to a string as we can ordinary CORBA objects. In other words, they cannot be referred to by an IOR. As a result, we store the structure containing the both the ORB and the BOA in a static/global C variable.⁴ This also means that we must provide interfaces to all of the methods that are supplied by these two objects and we cannot use the dynamic CORBA mechanism below to invoke them. Note that $\hat{\Omega}$ avoids this entire issue because the native objects can be stored and accessed in user-space.

The details of the contents of the command line arguments are ORB-implementation specific. See the document for the version you are using. Orbacus provides several flags and one which specifies a file containing properties. I recommend the use of that when possible. The IOR of the interface repository unfortunately cannot be specified there.

In general, one cannot re-initialize an ORB or BOA and be guaranteed the new configuration will be in effect. In other words, once you initialize these objects, you cannot alter the default settings taken from the command line arguments, configuration files, etc.

4 Locating CORBA Servers

Before we can invoke a method on a CORBA server, we must first locate it and obtain a “reference” or handle to it. This can be done in many ways and is the bootstrapping operation. CORBA provides two relatively simple mechanisms to refer to a server. One is to obtain its unique ID which is termed the IOR - the Interoperable Object Reference.⁵ This is a (long) string something like

```
IOR:0000000000000001f49444c3a537461744d61747269782f54726163655365727665723a312e3000000000
```

This can be read from a file, a web-site, or any stream and converted to a string. The ORB is capable of converting it to a C - or native-level reference to the object which can be used to invoke methods, etc. Because this type (string) is a user-level type in S and R , we will represent a CORBA object using its IOR as a string. Rather than treating an IOR as a simple string to be confused with other strings, we can give it a class of its own which extends string.

5 The Naming Service

A second mechanism for locating an CORBA server is more in keeping with the way we usually work in these (S and R) environments. The CORBA Naming Service is a hierarchical/nested container which stores references to CORBA objects using regular strings as names or keys. It is CORBA server in its own right and hence several applications can communicate with it. In this way, the names are shared across the different applications. Since it is hierarchical, different applications can create their own sub-namespace to avoid conflicts and ambiguity.

Given the ORB (and BOA) initialization, we can obtain the top-level naming service context using the user-accessible C routine

```
void R_CORBANamingService(char **obj, long *length);
s_object *S_CORBANamingService(void)
```

The return value is a single string whose contents is the IOR of the top-level object.

We can define subsetting [functions which access elements in the naming service. For example,

³We may not implement the S version since they are so similar.

⁴Of course, we will have to deal with this if we ever have threads and/or multiple ORB connections.

⁵It is Interoperable because the same IOR works across different ORBS.

```
> n = namingService()
> n["server"]
IOR:0003231....
```

⁶ Similarly, we can perform nested lookups using something like

```
> n["Folder1", "WithinFolder", "server"]
```

We could leave the possibility open for returning multiple objects and being more general so that this would return the three elements contained in the top-level naming context. In this case, we would extract the element named “server” located within the second level naming context using a syntax

```
> n["Folder1"][ "WithinFolder" ][ "server" ]
```

This allows us to have multiple values returned by specifying more than one names within a subset operation. So this is what we will do. However, there are still ambiguities that need to be worked out. For example, suppose the result of `n["Folder1"]` is a CORBA object but not a naming service, however it does support the notion of subsetting. How can we take advantage of this at the user-level. More on this later.

6 Invoking CORBA Methods

At this point, we should have enough to obtain a reference to one or more CORBA servers. Next, we want to utilize them by invoking methods they support. User-level access is via the function `.Corba()` akin to the `.C()`. It is used in the form

```
.Corba(server, "remote-method-name", args)
```

and is defined in the language something like

```
.Corba = function(server, methodName, ..., wait=T) {
  # potentially convert any IOR's to strings in the arguments.
  args = list(...)
  wait = as(wait, "logical"),
  .C("CORBAMethodInvocation", as(server, "character"),
    as(methodName, "character"),
    args # however these go down!
    length(args),
    wait, length(wait)
  )
}
```

This is charged with

- converting the IOR of the server to a C-level reference,
- computing the structure of the request as expected by the server,
- populating the request structure by converting the arguments to CORBA compatible values,
- converting the return value when it is

The last step is only done if the argument `.wait` is a logical vector of length 1 containing the value `true`. There are two other possible invocation styles. If the argument `.deferred` is specified, it identifies a list of requests. This CORBA request is stored in this list, potentially with other requests that can be considered a group or overall transaction. We can then return and query whether the request has completed or is still pending, obtain the output value(s).

The simplest invocation is a one-way request which just sends the instructions to the server. No return value is ever received and no out arguments are handled. This can be specified by passing the value **F** for the argument `.wait`.

⁶We can establish a better print method for the IOR objects.

`false` indicating that the call should not wait and that we are never interested in the return value(s). This is a one way call.

`NULL` indicating that the call should be a one-way call for which the user is not interested in the status.

The second and third steps require some machinery. To obtain the template structure of the request, we must query the object for the definition of the method of the given name. This is maintained by the object via the Interface Repository. This returns the IDL-based definition of the operation in the form of a Request which identifies the parameters and their types as well as the return value type. Given the former, we traverse the argument list and convert to the corresponding values. Note that we can do argument name matching at this point, as in *S* and *R*. This is not possible in *JavaTM*.

The argument conversion knows how to convert *S* primitives to CORBA primitives (and vice-versa for handling the return type). Additionally, it can convert IOR objects to their *C*-level CORBA references. When the return type is a CORBA object and not a primitive, it is converted to its IOR and that is passed back to the user.

7 INOUT and OUT IDL Parameters

IDL requires that each parameter of an operation be specified not only with a type but also an qualifier indicating whether the operation returns any information through the value of this parameter. Arguments that contain no return data are qualified with **in**. Parameters that are used only to provide return information are specified as **out**. Finally, parameters that are used in both directions are termed **inout**.

From our perspective, parameters that contain return information must be handled specially. For operations with such parameters, we arrange to return a list of values containing the out arguments and the return value. We use the parameter name from the operation to identify each element and the string `return` for the actual return value. If the operation is declared as **void**, the converter may leave the result as its default initialized value.

It is not really necessary to specify parameters declared as **out** in a call. The `.Corba()` can match argument names in the `...` with the IDL operation parameter names. This is now supported. The addition greatly complicates the `.Corba()` function and associated routine dues simply to the fact that we cannot allocate vectors in user-space for the return arguments since we don't know how many there are.⁷

```
> l = .Corba("Server", "test", m=matrix(1:9,3,3))
> names(l)
[1] "x"      "y"      "m"      "return"
> l
$x:
[1] 101

$y:
[1] 3.141593

$m:
[1] "IOR:010000001d00000049444c3a6f6d672e6f72672f434f5242412f4f626a6563743a312e30000000"
[2] "IDL:StatMatrix/Matrix:1.0"

$"return":
[1] 21
```

8 IDL Attribute Access

This mechanism can determine whether the target “operation” refers to an IDL attribute rather than an operation. In this case, it will attempt to get or set depending on whether there are 0 or more arguments respectively the value of

⁷We could do the argument matching in user space by incurring an additional call to obtain the operation characteristics. This should be available anyway. However, we run into the same problem for that function since we don't know the number of parameters for the operation (or the exceptions).

the attribute in the server. It does this by prefixing the requested operation with one of the 'set' and 'get' strings. This gives full access to server attributes.

```
> .Corba("Server", "count")
[1] 13
> .Corba("Server", "count", -213)
> .Corba("Server", "count")
[1] -213
>
```

9 User-level CORBA Arguments

When a user level object is supplied as the value of a parameter that is not a CORBA primitive type, we must do something special. In this case, we create a dynamic CORBA server using the DSI - Dynamic Skeleton Interface mechanism. It works like this. Since we know the expected IDL class of the argument from the parameter definition in the request, we simply create a proxy object that declares that it implements that IDL interface. This is a full CORBA object, but one for which skeletons have not been compiled and linked into this application. The proxy is then passed to as the value of the argument in the request. An additional vital step (which is not CORBA-implementation independent) is that we make this object available to receive CORBA requests as a server. This is done in Orbacus using the `init_servers()` method of the Orbacus BOA. This is an entirely generic mechanism for handling any non-primitive object for which we do not have stubs. The CORBA implementation (at least the Orbacus version) does not block when it dispatches the request to the remote CORBA server. Instead, it enters a local event loop waiting for CORBA communications of which a return value is just one. This allows intermediate requests to be passed to the arguments that were converted to CORBA servers (and other longer-lived servers in the application). It is at this point that we have to own up to the claim that was made for the user-level object and its ability to implement the IDL interface it is masquerading as.

When a CORBA request is received by the dynamic CORBA object or proxy object, it contains the same information that we constructed for our dynamic client request. Specifically, it contains the name of the method and the arguments. Firstly, we convert the arguments using the reverse mechanism as employed to create the request. CORBA primitive objects are converted to their user-level counterparts and CORBA objects are converted to IOR objects. Note that we potentially have the names of the arguments so we can create a named list of the arguments. Next, we take the name of the method and assume that it is the name of a user-level function. So we construct a call to this function parameterized with the converted arguments and evaluate it. The return value is then converted to a C-level CORBA compatible object consistent with the operations definition from its IDL interface. Finally, it is returned to the caller (by inserting it back into the request object).

10 Error Handling

Error handling is a thorny issue here. Having passed control back to the user-level evaluator, any errors there must be trapped by the caller of the sub-`eval()`. This is a the C-level and so the CORBA request can be terminated by inserting an exception into the callers request structure and returning it. The trick that I am not certain how to perform is the trapping of the error locally. If *S* and *R* had exceptions, this would be a simple exercise. The error in the evaluator would raise an exception that would ordinarily be caught only in the top-level parse-eval loop. By intercepting it, we could proceed graciously, allowing the original call from within our environment to possibly succeed because the original server in our call might handle the exception we threw!

In *S*, we can use the restart function. Since we pass the CORBA arguments, method name, etc. to an intermediate user-level function (by default `.CorbaMethodEval()`), we have it trap errors and return a suitable value. It can do this using the `restart()` function. We define a default argument (`errorOccurred`) for this function and check whether its value is structurally different from its default value. When we first enter this function, we perform the check. The first time through this check fails as the default value is present. So we proceed and evaluate the actual body of the code of interest. First, we change the value of this argument/variable. If the next segment of code generates an error, the function will be reinvoked due to the call to `restart()`. At this point, the check for a difference between the argument

and its default value is true and we return an object indicating an error. We do this by returning a list of length 2. The second element contains information about the error. If the function returns without signalling an error, the return value is a list of length 1 containing just the return value.

When we do discover an error in the DSI, we catch this using the above technique and the return an exception to the caller. The current implementation provides a method in the Interpreted CORBA Server class that creates the exception object. This is defined in the Omega IDL module in `OMEGA_HOMEInterfaces/CORBA/`. This currently provides two attributes that provide the error message from the server's system and potentially the function in which the failure. This will probably change and provide different information that is available. It is intended to be generic to apply to $\hat{\Omega}$, S , etc. Recall also it can be easily extended at the IDL level.

Notice that unless the IDL operation being called declared that it potentially raises such an error, in Orbacus at least, it will be reported as an exception of UNKNOWN type/class.

11 Conversions

The code implementing this general mechanism is itself very general and actually quite short because of this. Basically, it doesn't do much. Most of the code is concentrated on converting objects between the native system (S , R , etc.) and CORBA. These are responsible for converting primitives and importantly to and from non-primitive types. As mentioned above, the default converter for such objects creates a dynamic proxy object which uses the evaluator to implement all the methods by invoking the function with the same name as the IDL operation. There are two occasions where this may be undesirable and the user may wish to override this default converter.

The first reason to want to override the converter is to map different CORBA operation/method names to different user-level functions. For example, the IDL operation

```
data(in long, in long)
```

defined in the StatMatrix/Matrix example can be easily implemented in S/R as

```
server[i+1, j+1]
```

We can of course create a function or method for the function *data()*. However, this pollutes the name space. Instead, we may want a function to perform this mapping for us. Rather than use the default *.CorbaMethodEval()* (which uses the *do.call()* using the IDL operation name), another function can declare locally or lookup the name *data* in a table and use the corresponding expression or function.

A simple function such as

```
ExtendedCorbaEvaluator =
function(corbaArgs,
        methodTable=list(data=Quote(CorbaServer[i+1, j+1]), ..))
{
  e = methodTable{\Tt{}}corbaArgs[[1]]
  if(is.na(e)) {
    do.call(corbaArgs{\Tt{}}1, corbaArgs{\Tt{}}2)
  } else if(is(e, "call")) {
    eval(e, corbaArgs{\Tt{}}2) # need to add the CorbaServer
  } else if(is.function(e)) {
    e(corbaArgs{\Tt{}}2)
  }
}
```

or something vaguely similar does the dispatching locally or uses the default mechanism. One thing this does not do is handle errors correctly. Unless this uses an equivalent mechanism to trap errors, the default error handler will not

return to the C-level routine controlling the CORBA request. As a result, the remote CORBA caller will be left in a hung state waiting for an answer that it will never get. We may be able to set some error or signal handler.

Another mechanism is to create a new function that emulates `.CorbaMethodEval` but with functions defined locally. For example, the following is a direct copy of `.CorbaMethodEval()` but with a local version.

```
.Foo <-
function(els, interactive = F, errorOccurred = F)
{
  data <- function(m, i,j) {
    print("in internal data")
    m[i+1,j+1]
  }

  if(!identical(errorOccurred, F)) {
    warning(paste("error occurred in .CorbaServer", els{\Tt{}}1))
    return(list(NULL, error.level()))
  }
  restart(T)
  # make sure we change the the value.
  #
  errorOccurred <- T
  if(!is.null(els{\Tt{}}length(els))) {
    args <- vector("list", length(els{\Tt{}}2) + 1)
    args{\Tt{}}1 <- els{\Tt{}}length(els)
    args[2:(length(els{\Tt{}}2) + 1)] <- els{\Tt{}}2
  }
  else {
    args <- els{\Tt{}}2
  }
  # Names are bad news!
  # names(args) = els{\Tt{}}3
  val <- do.call(els{\Tt{}}1, args)
  ans <- list(val)
  return(ans)
}
```

In R, we can perhaps do this simply with a closure due to the scoping rules.

```
.Foo <- function(els) {
  data <- function(m,i,j) {
    m[i+1,j+1]
  }
  .CorbaMethodEval(els)
}
```

This has problems with the scope. However, we can do something like the following (see **R/matrixHandler.R**)

```
handlerGenerator <- function(this)
{
  setElement <- function(i,j, v) { this[i,j] <- v }
  getElement <- function(i,j) { this[i,j] }
  env <- environment()
  evaluate <-function(els)
    eval(substitute(els), env=env)
```

```

# this <- function() {
#   environment()$get
# }

# Don't need to package up the environment.
# env=environment()
return(list(evaluate=evaluate, setElement=setElement, getElement=getElement,
            this=function() { return(this)} ))
}

.CorbaServer("IDL:Omegahat/Matrix:1.0", Rmatrix=matrixHandlerGenerator(matrix(1:25,5,5)))

Client:

.Corba("Rmatrix", "ncol")
[1] 5
> .Corba("Rmatrix", "nrow")
[1] 5
> .Corba("Rmatrix", "getElement", 1,1)
[1] 1
> .Corba("Rmatrix", "setElement", 1,1, 100)
NULL
> .Corba("Rmatrix", "getElement", 1,1)
[1] 100
>

```

This setup still pollutes a local name space as we may need to distinguish between different same-named operations within different IDL servers. For example, we may have another IDL interface, say *Array*, which defines an operation *data*. It may take different arguments than the *data* operation above, or not. If we define methods for the function *data()* for the different types of servers, the interpreter will take care of this for us. However, if we want to avoid declaring such methods, we can specify such information in a slightly different form of *methodTable*. We can use the name of the IDL interface of the CORBA server as a key in a two-level table. The toplevel keys are the names of the IDL interfaces; the elements of the list for a given IDL interface are the same as in the example above - expressions or functions.

```

methodTable=list("IDL:StatMatrix/Matrix:1.0"=list(data=Quote(. . . . .)),
                 "IDL:Array:1.0"=list(data=Quote(other-expression)))

```

In order to have a specific user-level function be invoked in response to a CORBA request of a proxy server, we must pass this function name to the CORBA proxy object when it is constructed. Since this is done at the C-level, we must register the function ahead of time (compare with $\hat{\Omega}$). This applies only to entire IDL interfaces rather than per-operation. Hence, we require a simple one-way table which is indexed by the IDL interface name and contains simple strings identifying the name of the user-level function to employ.

The function *addCorbaConverter()* can be used to specify the user-level function to pass the CORBA arguments for a proxy server request. The arguments are the name of the user-level function, the name of the IDL interface as specified by the IDL compiler and used in regular lookups in the Interface Repository. To indicate that this is a user-level function to be used for proxy servers of this IDL type, one specifies **T** for the *userFunction* argument. This is necessary since the function also supports the specification of native routines to be used for converting to or from CORBA objects. This is done by specifying the name of the routine (as it would be passed to *.C()*) as the first argument and specifying either **T** or **F** as the third argument, *toCorba* indicating whether it a converter going from user-level objects to CORBA or vice-versa. As usual, the user-level system will assume the routine has the appropriate declaration of type *fromCORBAConverter* or *toCORBAConverter* (defined *GenericConverter.h*)

```

> addCorbaConverter("createCorbaMatrix", "IDL:Omegahat/BaseMatrix:1.0", userFunc-
tion = T)
{\Tt{}1}:

```

```

[1] ""                                "IDL:OmegahatMatrix/BaseMatrix:1.0"

{\Tt{}2}:
[1] -1

> addCorbaConverter("duncan", "IDL:Omegahat/BaseMatrix:1.0", userFunction = T)
{\Tt{}1}:
[1] "duncan"                            "IDL:OmegahatMatrix/BaseMatrix:1.0"

{\Tt{}2}:
[1] 1

> .Corba("Server", "trace", matrix(1:9, 3, 3))
Using default converter to CORBA for IDL:Omegahat/Matrix:1.0 duncan
[1] 15

```

To register a native symbol to convert `IDL:OmegahatMatrix/Matrix:1.0` to a CORBA object, one can define a routine with a special signature and then connect this with the appropriate IDL interface via the converter registration. We provide a basic converter for *Matrix* named `createCorbaMatrix()` (see **BasicConverters.nw**). It is registered something like

```
addCorbaConverter("createCorbaMatrix", "IDL:Omegahat/Matrix:1.0")
```

This then uses the native code to implement the object. The programming facilities user are available as macros in both *S* and *R* and make this more feasible than previous versions of these environments. This is akin to programming in routines used with `.Call()`, specifically we are operating on the native/internal objects used in these systems.

The information about the current set of converters can be obtained using the function `getCorbaConverters()`.

Note that these are maintained in memory and are not automatically read when the chapter is attached. This is of course relatively easy to implement as the `addCorbaConverter()` can maintain a persistent object that is synchronized with the memory version and reinstantiate it during the `.on.attach()`.

The native converters for both

Native routines must have appropriate signatures/prototypes. These are typedef'ed in **GenericConverter.nw**. A converter from a CORBA object to a native one

Soon we will allow functions and closures be registered as converters for converting to a CORBA object. These will be used for dispatching requests in a proxy CORBA object that sits on the CORBA bus. Native routines are different and create the actual CORBA object that sits on the bus. How the requests it receives are dispatched is up to it.

The performance improvement when using natively implemented rather than dynamically interpreted is significant when we use these natively implemented CORBA servers. The time spent is composed of network transfer time, interpreter time, etc.

One of the interesting (and unexpected) results that we obtained during very early and casual experiments was that, in *S4*, when contrasting performance for the dynamic and native implementations, we ran into a copying problem due to the natural implementation of the *LocalMatrix* class. The call to **AS'NUMERIC** to convert the matrix to an array of *doubles* exceeded the maximum size of an object. This illustrated the fact that the data was being copied and that matters were slightly more complicated than we might imagine.

On the other hand, the results below are a stunning indication of why we want to implement the servers in *C++*. I will take the opportunity to point out that $\hat{\Omega}$ will provide a compiler which will allow the interpreted version to be compiled in pure *JavaTM* code and potentially into machine code (using the Cygnus *JavaTM* compiler).

We can continue this investigation by extending the IDL interfaces *Matrix* and *TraceServer* to reduce the number of CORBA calls and hence interpreted requests. We add a method `allData` to *Matrix* in the interface *ExtendedMatrix*. This operation returns a sequence of doubles containing the contents of the entire matrix. Then we define an operation named `fastTrace` in an interface *ExtendedTraceServer*, an extension of *TraceServer*. This calls the `allData` and computes the dimensions via CORBA calls to its sole argument. After that, it computes the actual value of the trace locally.

We implement this via extension in the IDL sense and implementing the new IDL classes by extending the existing ones.⁸

Note that when we transmit the contents of the matrix, we have to create a large array. Depending on the size and the configuration of the JVM, this may exceed the available memory. While this may seem a limitation, it is really a different configuration. We have many choices as to what services we can provide on a given machine. For hardware with little RAM, we can offer the slower version of *trace*. This utilizes a lot of network bandwidth, but little memory. Where a single but lengthy communication is better and large quantities of RAM are available, the speedier *fastTrace* may be preferable. Of course, the decision also depends on the client(s).

The following times are given as User, System and Real.

Dynamic Interpreted proxy server.

```
> m = matrix(1:1000000,1000, 1000)
> unix.time(.Corba("Server", "trace", m))
[1] 26.63 23.98 54.15
> unix.time(.Corba("Server", "trace", m))
[1] 25.88 24.35 54.27
> unix.time(.Corba("Server", "trace", m))
[1] 27.38 22.80 53.77
```

Native (C++) Implementation CORBA Server. This has no interaction with S when communicating the matrix attributes to the server other than the initial request in the *.Corba()*.

```
> addCorbaConverter("MatrixConverter", "IDL:StatMatrix/Matrix:1.0")
{\Tt{}1}:
[1] "" "IDL:StatMatrix/Matrix:1.0"

{\Tt{}2}:
[1] -1
> unix.time(.Corba("Server", "trace", m))
[1] 0.35 0.19 1.26
> unix.time(.Corba("Server", "trace", m))
[1] 0.37 0.10 1.41
> unix.time(.Corba("Server", "trace", m))
[1] 0.31 0.20 1.40
>
```

Fast Native implementation of Matrix server and using *allData()*. Large cost of sending data *once*. Note that to handle this in the *JavaTM* server, we must specify a suitable value for the *-ms* argument for the *JavaTM* virtual machine.

```
> addCorbaConverter("ExtendedMatrixConverter", "IDL:StatMatrix/ExtendedMatrix:1.0")
Function symbol ExtendedMatrixConverter 0x4101d8e4
[1] "" "IDL:StatMatrix/ExtendedMatrix:1.0"
[3] "-1"
> .Corba("ExtendedServer", "fastTrace", matrix(1.:9.,3,3))
[1] 15
> options(object.size=10000000)
> .Corba("ExtendedServer", "fastTrace", m)
[1] 500000500
> unix.time(.Corba("ExtendedServer", "fastTrace", m))
[1] 0.27 0.40 12.06
> unix.time(.Corba("ExtendedServer", "fastTrace", m))
[1] 0.25 0.21 9.50
> unix.time(.Corba("ExtendedServer", "fastTrace", m))
```

⁸Due to the lack of multiple inheritance in *JavaTM*, this is slightly more involved than in *C++*. For the moment, we use the noweb facilities for repeating code. However, we can use the CORBA tie mechanism to use delegation.

```
[1] 0.27 0.16 9.61
>
```

Dynamic user-level (slow) implementation but using the *fastTrace* and hence the single transfer of all the data. Note that the real times are decreasing here!

```
> options(object.size=10000000)
> .Corba("ExtendedServer", "fastTrace", m)
S Proxy object for IDL:StatMatrix/ExtendedMatrix:1.0
[1] 500000500
> unix.time(.Corba("ExtendedServer", "fastTrace", m))
S Proxy object for IDL:StatMatrix/ExtendedMatrix:1.0
[1] 0.57 0.66 15.24
> unix.time(.Corba("ExtendedServer", "fastTrace", m))
S Proxy object for IDL:StatMatrix/ExtendedMatrix:1.0
[1] 0.60 0.57 14.70
> unix.time(.Corba("ExtendedServer", "fastTrace", m))
S Proxy object for IDL:StatMatrix/ExtendedMatrix:1.0
[1] 0.46 0.50 11.46
> unix.time(.Corba("ExtendedServer", "fastTrace", m))
S Proxy object for IDL:StatMatrix/ExtendedMatrix:1.0
[1] 0.55 0.38 10.90
```

So, the purpose of providing this interface is to help introduce CORBA to users by providing high level access within a language they already know. Additionally, it allows employment of servers locate dynamically for which the effort of compiling, linking and loading stubs is considered prohibitive. Basically, it allows users to decide on the relative tradeoffs between development and run-time costs rather than being forced to implement CORBA servers natively and linking back to interpreted languages. Additionally it makes certain connections feasible that would not otherwise be. Finally, I hope that once the basic CORBA mechanism is understood, it will be relatively easy to implement some of the user-level functionality in “more native” and object-oriented languages. The *LocalMatrix* example illustrates how simple this can be. The addition of the user-specifiable converters allow different classes to be implemented natively where this is a good use of resources and others to be implemented in the user-level language.

12 Implementing the Converters

Initially I resisted the use of classes to implement the basic CORBA facilities. As the number of environments, CORBA implementations and operating systems grew, the code became complicated and heavily reliant on a plethora of preprocessor macros to hide the differences between the varying elements. So, I switched to *C++* classes to implement the basic converter functionality.

One motivation for this was that we needed to store the CORBA objects created by converting user-level objects to CORBA objects in a *.Corba()* call. Since these must be removed from the CORBA bus at the end of the call, we need a namespace in which to store them. Since these are created during the conversion process, it seems suitable to store them in an object associated with the converter.

Also, many of the differences arising in the converters for the different environments are orthogonal to the CORBA interface. Instead, they focus more on how to convert primitives from user-level representations to primitive objects and vice-versa. Also, *R* and *S* share much in common when it comes these conversions, differing only in the handling of string vectors.

Accordingly, the basic mechanism from extracting from and inserting into CORBA objects is provided in a base converter class called *BaseCorbaConverter*. This just handles the identification of the source and target types in terms of primitives and sequences. It implements the conversions by calling virtual methods that are provided by derived classes.

Since *R* and *S* are so similar, we implement a common class derived from *BaseCorbaConverter* that implements most of the low-level

BaseCorbaConverter

RSCorbaConverter OctaveCorbaConverter

RCorbaConverter SCorbaConverter

13 C-level Servers

The dynamic invocation and server mechanism is extremely useful and even necessary in interactive dynamic environments like *S*. They dramatically simplify prototyping and experimenting with CORBA facilities, in much the same way that *S* removes the compilation stage of development. However, for most real applications, the IDL types are known in advance, due to careful design by the authors. In such cases, we can and sometimes should still use the dynamic facilities and implement the servers in the interpreted language. At other times, it will be convenient to use *C/C++*-level implementations of CORBA servers.

To override the default converters, we must again associate an object with an IDL interface. Here we have to opportunity to convert in both directions – to and from CORBA objects. In this case, the conversion must be at the *C/C++*-level. We provide **typedef**'s for these two types of converter routines – `fromCORBAConverter()` and `toCORBAConverter()`. The user can then register a *C/C++* routine to be used for a given IDL interface using the user-level function `addCorbaConverter()`. At the moment, an exact match of the IDL name and the key used to register it for a converter to take effect. This means that derived interfaces will not use this until we implement the search for super-classes.

This discussion does not apply to using *C/C++* stubs in the interpreted language since that can be done trivially using the `.C` mechanism. One would usually refer the object at the user level using its name from the naming service or its IOR and pass this to a *C* routine which converts and narrows the object appropriately and invokes the corresponding object. One can use the standard converters to convert the arguments and return value.

```
trace =
function(m, corbaServer) {
    storage.mode(m, "double")
    .C("S_corbaTrace", corbaServer, as.numeric(m), dims(m), 0.0)
}
```

The implementation of the *C* routine can be something like below. We use the `stringToCorbaObject()` which converts a name to a CORBA object. We suppose the the actual IDL class of the server is `MyClass` so we use the stubs to narrow the generic object to this class. At this point, we can invoke the method `trace()` as if it were local. The argument to this CORBA method is a matrix for which we may have the skeletons and a local *C++* implementation named *CorbaMatrix*.

```
void S_corbaTrace(char **ior, double *vals, long *dims, double *ans) {
    CORBAObject_ptr o = stringToCorbaObject(ior, LocalCorbaGlobals);
    MyClass *server = MyClass::_narrow(o);

    ans[0] = server.trace(new CorbaMatrix(vals, dims[0], dims[1]));
}
```

To use the default converters to create the Matrix, we would pass the user-level matrix as a regular object as an element of a list and the use the `S_toCORBA()`.

```
void S_corbaTrace(char **ior, void **m, double *ans) {
    CORBAAny_ptr any = new CORBAAny( , (void *)NULL);
    S_toCORBA(any, (s_object*)m[0])
}
```

14 Deferred Calls

Deferred calls can be used to group several related commands into one CORBA dispatch. Why is this necessary - because none of the languages we address here have support for threads. The idea here is that we create several future calls to CORBA servers and buffer these to be called in one action. The benefit of this is that the dispatch of the requests can then block waiting for all to respond in some manner, leaving the evaluator unoccupied.

Consider the scenario that we have k calls to be dispatched to CORBA servers. For simplicity, we will use a for-loop to define them, however this is not a requirement. The commands can be given explicitly in order. A simple example is that we have k identical servers and we store their names in the character vector `servers()`. Additionally, we have a list of the same length whose elements identify the observations to be used in this sub-computation. These are the indices of this sub-group of observations. So the following

```
for(i in 1:length(servers)) {
  .Corba(servers[i], "subComputation", indices{\Tt{i}},
        deferred = "subComputation", send = F)
}
sendCORBARequests("subComputation")
```

arranges to create the requests and then dispatch them all. The loop creates the calls in the usual way. The last two arguments however arrange to not dispatch the call but to store it in a list which can be accessed. Note that this is not directly accessible since it is stored in a C structure. It can be manipulated using the different deferred request methods. The `sendCORBARequests()` is one such function. This is used to dispatch the calls. In the example above, all the requests in the list indexed by "subComputation" are sent. Subsets can be specified. Similarly, the dispatch can be done as blocking, non-blocking or one-way.

How do we get the return values? There are several ways, depending on how the calls were dispatched. Firstly, we can use the `getRequestValue()` function. This, and the `getRequestStatus()` function, "polls" the request. In other words it checks whether it has finished when the caller decides to check. It can be set not to block but still polls.

A second way is to avoid polling and instead be notified when the task is complete. When we define the IDL operations, we can arrange to have this happen. We pass an additional argument to the operation which is to be notified when the server has completed the sub-task. This object must have a particular known method that is invoked by the server, say `completedTask`. Whether this returns the result, identifies any errors or merely identifies itself and the task is up to the IDL author. The original request is still present and the results or exceptions can be extracted from it. (This assumes it was not a one-way call!)

Another representation for inputs to the example above is to use the server vector as the names of the indices list. This suggests (but does not enforce) that we have linked the server to that group of indices. In some cases, this may be ideal and we may want to arrange that a particular server maintains its own copy of a subset of the data across operations. This form of caching can lead to dramatic reductions in transmissions and essentially provides static variables in this distributed setup.

When the tasks vary in nature and duration, refer to different data or the servers are located on hardware with very different performance characteristics, it is often useful to be able to dispatch k' of the total k tasks, where $k' < k$. In this way, we can "schedule" the tasks in different ways. We can dispatch these to different servers based on the nature of operation (e.g. speed of hardware, network traffic/bandwidth access, existing data on the server, expected duration of the operation, etc.) As one task is completed and returns to the original caller, the next task can be dispatched. This can go to the now-idle server or perhaps to one drawn from a pool of potential servers.

Deferred calls are a bad idea where arguments are user-level CORBA servers and we are in the non-threaded world. The reason for this is that we have passed control back to the evaluator immediately from the `.Corba()` call. However, the newly created user-level CORBA servers are still in effect. When a request is handled by one of these and passed to the user-level function, the evaluator is reentered. Either this will block while the evaluator is being used or else it will start up the evaluation in a potentially bad point in the evaluator. This will usually lead to a crash of some sort.

```
> .Corba("ExtendedServer", "back", 2.12, .send=F, .deferred="A")
[1] 0
> .Corba("ExtendedServer", "foo", "my string that will be deferred", .send=F, .deferred="A")
[1] 1
> sendRequests("A")
```

```
[1] 2
> getRequestValue("A", 0)
$"return":
[1] 2.12
```

```
> getRequestValue("A", 1)
$"return":
[1] 31
```

15 Toplevel Servers

We have seen how CORBA objects are automatically created in operation calls. We do need a mechanism however of creating CORBA servers directly and putting them on the CORBA bus for clients to access. For this, we use the `.CorbaServer()` function. This takes the name of the IDL interface which the new object will implement. The usual converters outlined above are used to create the CORBA server. By default, a dynamic interpreted server is created which passes the requests to a user function. This function can be specified as the *4th* argument.

The second argument is the local object being put on the CORBA bus. This contains the data for implementing the different operations.

The third argument is the name(s) which the new object will be bound to in the naming context. This is a character vector identifying the naming context into which the object will be bound and the name to which it will be bound.

The fifth and sixth argument control how the object is made available to the CORBA bus. If the first of these, *block*, is **T**, then the call immediately makes the new CORBA object available and does not return until it is deactivated (or interrupted).⁹ If this argument is **F**, then the second of the arguments *activate* governs when the object is made publically available. If this is **T**, then the object is immediately made available. However, since *block* is **F**, the call returns. This is dangerous if done at the top-level and should only be done if you *really* understand the computation model involved here. If the argument is **F**, then the object is just created but is not ready to receive requests. This might be used to create several objects in succession and then make them all available when the last is created.

A simple example involves two *S* processes. In the first, we create a simple matrix that implements the *Matrix* interface used earlier. We make it available under the name `smat`. The basic dynamic server is used and the methods `- nrow`, `ncol`, `data` - by passing the request data to `.CorbaMethodEval()` which takes care of calling the corresponding functions.

```
> .CorbaServer("IDL:StatMatrix/Matrix:1.0", matrix(1:9,3,3), "smat")
```

In a client process, the following commands illustrate how we access the remote server in an identical fashion to earlier CORBA servers. Keep in mind that we are invoking the IDL methods, not the *S* functions in the server process. The fact that the names are the same is a result of the server being implemented in the simplest possible fashion.

```
> .Corba("smat", "data", 0, 0)
[1] 1
> for(i in 1:.Corba("smat", "nrow")){print(.Corba("smat", "data", i-1, i-1))}
[1] 1
[1] 5
[1] 9
>
```

This function has now been vectorized so that one can create multiple servers in one command. Because of the different values one can specify, this becomes slightly complicated to describe. Basically, one can specify multiple names. These must be provided as a *list*. This allows nested names to be specified as character vector elements within a list. Similarly, one can specify multiple objects. Again, these are supplied as elements of a list. The list of names and objects should have the same length unless one is omitted. The IDL types are repeated to the appropriate length.

At present, if one specifies multiple names, things do not work. There is a problem with the registration of the objects with the naming service.

⁹We need to handle signals here to kill the object!

16 Events

An alternative to deferred calls is to use CORBA events. Since *S*, etc. are not threaded, we use a simplified version of the CORBA event model. Specifically, we use the non-automatic notification models. Instead, the source of the events notifies the CORBA event channel. Similarly, consumers of the events decide when to retrieve events from a channel.

Like deferred calls, event channels are identified by name.

17 Dynamic Extension

The interpreted DSI and DII approaches allow us to do some interesting things not possible with compiled skeletons and stubs. Suppose we have an IDL interface, say *A*. If we decided that it was underspecified and requires another operation to be added to it, we can do this via extension in the IDL language. Then we can compile stubs and skeletons for the new interface and proceed to implement, compile and load them into our applications. However, this requires that we destroy the existing server objects and client applications and reconstitute them. The servers will now implement this new IDL interface and the restarted clients can call the new operations.

Using DII, the clients do not need to be restarted in this case. Since the lookup of an operation in the IDL interface is dynamic (even if cached), the new operation will be located and can be used as if it had been there at all times. The only thing we need to have done for this is to feed the modified IDL interface to the Interface Repository.

If servers are implemented using an interpreted DSI implementation, we need not reconstitute these either. Instead, the new IDL operation is immediately available from the server to any (DII-based) client. A request to the new operation is propagated to the interpreted world and resolved there. The maintainer of the system can define a function or method to handle this new operation. This definition can be after the new operation has been added the IR. Indeed, the beauty of the dynamic interpreted servers is that we can modify the implementation of the requests in the middle of a request. In *S*, for example, an error in a user-level function which is used to implement the CORBA server may give rise to a debugging browser. This may allow us to dynamically alter the run-time behavior of the function and make the necessary repairs for future calls.

In order to make such changes to an IDL interface, we might edit the source file and then re-run the equivalent of the ORBacus `irfeed` command. In many implementations, this will not work as it will complain about redefinition of an existing interface. Instead, we could restart the IR and repopulate all of its contents. In a long running server, this may be a managerial problem. The ORBacus utilities include an `irdel` which allows entries to be removed to avoid this problem.

An alternative and more user-friendly approach is to programmatically add attributes and operations from within the user-level environment. This is done using the `create`attribute` and `create`operation` operations available in an interface. Similarly, we can create new interfaces, aliases, sequence types, etc. from a non-source based mechanism. When combined with the IRViewer tool in $\hat{\Omega}$, this is another mechanism to construct and edit the repository. See `EditableIDLInterface.jweb` in the $\hat{\Omega}$ distribution.

18 Evaluation & Thread Models

Several CORBA implementations provide the facilities to handle requests in any of the following ways

- serially,
- one-per-thread
- thread pool

Our model must be very careful which one it employs. The first will work in all cases. $\hat{\Omega}$ has the capabilities of handling the latter two since it is designed to be a thread-safe evaluator. In implementing the user-level functions however, one must be careful to preserve the thread-safety.

19 Code Layout & Files

Since we are supporting at least two systems, *S* and *R*, we have to code that works for both and their derivatives. We also want to have the same user API for both systems.

The idea is that we have the basic *C++* routines that do the work and then the system-specific wrappers that the user can call to access these other functions. This is not just a simple matter of identifying the symbol (i.e. name mangling avoidance) but also of passing and converting the arguments correctly.

Most of the *C*-level code is in fact *C++* and requires a suitable compiler. We use the egcs on Linux and CC on Solaris.

The initialization of the ORB and BOA is in **CORBAinit.nw**. This also stores the structure containing the initialized values.

The naming service access is provided in **NamingService.nw**. This provides the facilities for obtaining the naming service

The directory **UserCode/** contains the user-level functions. This contains the code for all versions of the environments and will likely use Brian Ripley's perl script in conjunction with Doug's (and mine) GNU make rules to generate version-specific files from the single source.

The second-tier configuration files that parameterize many of the CORBA types for generic referencing/use across different CORBA implementations (Mico, Orbacus, Visigenic to name the three I have used) are in $\$OMEGA_HOME/Interfaces/CORBA/$. Hence you *need* a (configured) version of $\hat{\Omega}$. Since it has several useful tools for viewing the CORBA environment, there is additional value to having it.¹⁰

20 S Classes

It seems highly reasonable to only refer to objects by name and their IDL interface. So as well as its IOR or name, we should be able to tag on the return value from the object's method `_ids()`

21 Examples

The following are examples from calling the servers implemented in the $\hat{\Omega}$ distribution. They are described in $OMEGA_HOME/Interfaces/CORBA/DynamicJavaObject/Test/$ and $OMEGA_HOME/Interfaces/CORBA/DynamicJavaObject/Docs/$ respectively.

```
> .Corba("SimpleServer", "random", 10000)
[1] 0.4240232 0.6140213 0.7478235 0.7468567 0.9194604 0.6611324 0.5356237
[8] 0.4846376 0.1036596 0.1293056
> .Corba("Server", "foo", "This is string")
[1] 14
> .Corba("Server", "foo", 10)
[1] 2
```

22 Target Systems

The goal is to have this code support a variety of different statistical packages. The current targets are *S* versions 4 and perhaps 3; *R*; *Xlisp-Stat* & *Matlab*. The **NamingService** may need to be tweaked for these depending on how the equivalents of the `.C()` function works and passes the return values.

The lack of a restart function in the current version of *R* needs to be handled.

I have defined some macros in the **CORBAConfig/** to take care of the differences between *S* and *R*. These include `COPY_TO_USER_STRING` maps to `R_alloc()` and `c_s_cpy()`.

R appears to store its character vectors as a collection of `USER'OBJECT'` pointers. Thus to get the *i*-th element, we look in the string slot of the union of the *i*-th element of the vector. This is different from *S*. There, the character

¹⁰In the future, I will make a condensed version of the necessary files that can be used independently of the $\hat{\Omega}$ source.

strings are stored contiguously and we extract the entire collection from the *S* object container in one go. Then we treat the vector as a simple *C*-level array.

In order to accomodate this difference, we pass the USER`OBJECT` as an argument where we would have passed the extracted the char `**`. Then, we use the macro CHARACTER`VECTOR`ELEMENT(*x*,*i*). For *S*, this incurs a fractional unnecessary expense of dereferencing.

I am including *S.h* from both `include/` and `src/include/` in the *R* source tree. However, both of these define (rather than declare) `R_problem_buf`. Hence, we get multiple definitions of this symbols when we link. In effect, this is supposed to be included by a single file.

22.1 Matlab

I believe the basic mechanism will port to *Matlab*. That system seems to have a well-developed *C* interface with the ability to call *Matlab* functions (`mexCallMATLAB()`).

23 Installation

The code has now been reorganized so that the different layers live in separate directories, etc. so as to maximize reuse and potentially simplify modifications, etc. When the material is installed, the different libraries and include files live in `“Env -OMEGA`HOME~/lib/` and `“Env -OMEGA`HOME~/include/` respectively. Accordingly, each user need only set their environment variable `LD_LIBRARY_PATH` to

```
.:$OMEGA`HOME/lib:/usr/local/lib
```

(where the use of `/usr/local/lib` is where the Orbacus libraries are installed and `ldconfig` does not necessarily resolve them correctly).

The *R* specific code is in `“Env -OMEGA`HOME~/R/CORBA/`. From that directory, the default GNU make target will take care of creating the appropriate libraries and installing them in the appropriate directory.

Note that these default dependencies are meant for installers who are not modifying the code. Those people that are recompiling modifications frequently must install manually from the directory within which they are recompiling.

When developing this library, I got myself into a horrible position. I had an installation of the then most recent Orbacus. However, because of the nature of the installation (across multiple Linux machines some of which did not have the latest egcs but were on the regular gcc (2.7.*)) I did not build the Orbacus shared libraries, but just the static ones. Then, the *C++* IDL compiler generated code for the new Orbacus libraries. However, when linking and loading, the old shared libraries were being used and some bizarre symbols were missing. Specifically, it was a casting operator for the class `TypeCodeConst`. Reinstallation of the Orbacus shared libraries fixed this, but it was aggravating!

24 CORBA Implementation

Mico is looking like the potential target given that it is GPL'ed. The current source compiles except for handling Any in the converter and exception handling. Finally, there is a slight problem with the need to redefin vector for *S*(4). This is not a problem.

We can easily communicate with different CORBA implementations and objects created within these different sub-systems. For example, the following *S* commands shows how we can obtain the contents of a naming context which is the ORBit naming service on a different machine. Note that the contents (the single nested NamingContext named `Duncan`) were created in $\hat{\Omega}$ using Orbacus.

```
ir = ior("/tmp/Orbit")
namingServiceContents(ir)
  Duncan
    T
```

Ideally, we would be able to interchange different CORBA implementations. However, given the current state of many of the free ORBs this is not possible. ILU and ORBit currently do not have facilities for DSI and partial support for DII. Obviously, in our case we need, the DSI to make the connection with arbitrary user level objects.

Note that using shared libraries with certain compilers (e.g. egcs 1.0.2) doesn't allow exceptions to be caught. This means that if you attempt to resolve a name that doesn't exist, the session will terminate!

25 Adding a New System

- Identify how to pass arguments to C routines.
- Converters to primitive types.
- Mechanism for calling user-level function from *C/C++*
- Memory management.

The *R/S* and *JavaTM* interface allows one to call *JavaTM* methods from *S* and vice-versa. This is, in some sense, a less general remote method invocation mechanism with several advantages. The first is that the communication is within the same process and hence faster. This is also a drawback as it doesn't get the distribution. It is also Java-specific and does not address the issue of communicating with other languages. While one can use *JavaTM*'s RMI mechanism, this is very different from CORBA as it involves non-lazy copying of data. One of the major advantages of the *S/JavaTM* interface is that it does not require the creation of IDL. This makes it easy to get started and call existing *JavaTM* classes. For simple applications, this is fine. For more ambitious software development, the creation of IDL provides significant benefits. Firstly, it helps one formalize the different components and their responsibilities in the environment. Secondly, it allows for easy substitution and replacement of components after the system is developed. This is the same reason we program against interfaces in *JavaTM* rather than actual classes.

18 *<GPL License 18>*≡
Copyright (c) 1998, 1999 The Omega Project for Statistical Computing.
All rights reserved.

<GPL License 18>