

CORBA - the Common Object Request Broker Architecture - is a powerful framework which allows software components to be used together. What makes it exciting and powerful is that the different components can be located in different processes and on different machines and can be implemented in different languages, such as *JavaTM*, *C* and *C++*. The different CORBA objects can be treated identically to local objects making their use transparent.

This is a very powerful mechanism akin to the UNIX pipe framework. It allows specialized tools to be developed and linked with other tools to perform tasks. It offers great possibilities for software in general, and definitely allows statistical environments to share services and become less monolithic and more adaptable.

CORBA *appears* complicated but this is due to the generality it offers as well as the fact that it typically made available in newer languages that we do not use in our daily work - namely, *C++* and *JavaTM*. The goal of the tools described here is to provide access to and the functionality of CORBA within the interpreted languages we use to do data analysis and research. These include *S*, *R*, *Matlab*, etc.

The functions presented here are not intended to form the final end-user API. Instead, they are the primitives that others can build upon in the interactive language to provide high-level access to the CORBA environment. However, they are already reasonably high-level and can be used immediately.

Future plans involve accessing other CORBA services (e.g. the property service, etc.) and implementing a version of the *For()* function which distributes and manages tasks with fault tolerance.

1 CORBA Basic: IDL

The heart of CORBA is the communication between client and server in the form of attributes and operations. In order for this to work, both client and server must agree on what the server can do and how to call it. The server declares its list of accessible attributes and operations as an Interface. This is specified using the Interface Definition Language (IDL). This is not used to implement the server (or client), just describe its capabilities.

A simple example is given below to illustrate the basic nature of IDL. Here we describe the facilities available from a Matrix object. (This is not intended to be complete. We will see how to extend this naturally later in the document.)

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

Note that each parameter has a qualifier that indicates whether the argument is either **in**, **out** or **inout**. An **out** parameter is used to transmit data from the server to the client. An **inout** argument is used to pass information from the client to the server as a regular argument (**in**) but also is used to communicate values back to the client from the server. The **out** arguments are used much like passing pointers in *C* so they callee can modify the contents of the object. It allows more than one object to returned, avoiding putting them into a container structure and making this the return type.

An IDL description of a server's capabilities is all that is needed to define the communication between a server and its clients. Now we implement these. Compilation of the IDL to a suitable target language is the usual manner of implementing either a server or client. Most CORBA implementations provide an IDL to *C++* compiler. Others also offer IDL to *JavaTM*. Since there are no IDL compilers for interactive languages such as *S*, *R*, etc., we have to create another mechanism. Fortunately, CORBA provides one which is termed *dynamic*.

The idea for the dynamic CORBA interfaces is that we store the IDL description of a server in an Interface Repository (IR). When a client attempts to invoke a method of a server, a generic mechanism fetches the description of the operation from the IR and puts the local arguments into that request in some manner and sends the request to the server. For its part, the server cannot tell how the client generated the request. It responds and puts the return value(s) into the request and transfers it back to the client. There, the generic mechanism unwraps the return value(s).

This generic mechanism provides direct access to the CORBA facilities from within a language without the need for compilation of client stubs from the IDL and their compilation in the native language (e.g. *C++*). We can use a second generic mechanism to allow user-level objects (e.g. an *S* matrix) to be offered as a CORBA server that other clients can access. Again, we wish to avoid compilation and to allow the implementation be done in the user-level language rather than the lower-level native language. This server mechanism creates a proxy CORBA object that contains the real user-level object. The proxy object declares that it can service requests from a particular IDL

interface. When it receives such a request, it converts the arguments and passes the call to an the appropriate user-level function (e.g. *nrow()* in our example above). when this has completed, it puts the return value back into the request and returns it to the caller.

See adding operations and attributes dynamically.

2 Identifying CORBA Objects

In order to avail of CORBA servers, one must first obtain a “handle” or reference to an appropriate CORBA object. Appropriate here means an object that supports the require operations and is configured or configurable to the needs of the caller, including access to data not passed as arguments of an operation, CPU cycles, etc. However, these are matters that are handled using different technology and management.

There are two basic ways to obtain a reference to a CORBA object. The most common mechanism is to use simple names. A server registers itself with what is called the Naming Service. This is a CORBA object that holds references to server objects. Clients can obtain the reference for a server using an agreed name. The agreement is between the server and all its clients to use the same name. To avoid name conflicts, multiple servers can be used, or more typically different sub-contexts can be created. This gives file-system like hierarchical or nested structuring. To make this more concrete, consider the setup our statistics department might employ. Each user may have their own area in which they can register servers for use primarily by them alone. We might construct a naming context named *duncan*. Inside this, I might place some persistent servers that I want be around for lengthy periods - e.g. datasets, plot servers, my own IR for IDL interfaces I use. I might also organize different projects in their own naming contexts within my own personal one. Other servers used by more than one person may be put in the top-level naming context or in a department area.

To refer to top-level objects, we use a simple string such as “*Server*”. As a general rule, to refer to nested name, we pass the components as a character vector of the form

```
c("A", "B", ..., "Server")
```

An *S*-specific syntax is described below which allows us to perhaps more naturally specify a nested name in the form

```
A$B$C$...$Server
```

The second mechanism for referring to a CORBA object is the Interoperable Object Reference (IOR). CORBA provides a facility for converting any server to a string which allows the reference to be obtained from it. The methods `object_to_string()` and `string_to_object()` perform this interchange between object and string. The result of the former can be passed via a file, or made available on a Web page. A client can then convert this to an object and use it as if it had been obtained from the naming service. This mechanism allows servers to be obtained when naming services are not shared. Of course, there are many security issues to deal with here.

While *S* does not provide a mechanism to read an IOR from an HTTP server, the function `ior()` is a simple function to read a single IOR from a file. This can then be passed as the argument to most of the functions that also accept a name to be resolved in the naming service.

If one uses Orbacus as the ORB implementation, etc. there is an additional way to specify a named object. This uses an Orbacus-specific extension which allows names to be of the form

```
iiop://host:port/name
```

(Note that at present, the code is very strict about this format.) This is similar to a URL specification. It is a convenient way to specify the naming service or interface repository (see below) running on a different machine or of a different user. See chapter 6 of the Orbacus manual for more details.

$\hat{\Omega}$ provides a tool to examine (and edit) the contents of the Naming Service.

```
omega -CORBA -ix NameServerViewer
```

Since naming servers are themselves CORBA objects, we can refer to them via an IOR or a name within a super-level naming service. While this is possible for individual operations (e.g. via the *server* argument to `namedCorbaObject()`), this is inconvenient for groups of calls or sessions that want to refer to a different default naming service. Accordingly, the function `namingService()` can be used to both obtain the IOR of the current top-level naming service

and set it to a different server. When we set the value, all subsequent calls that use the naming service will communicate with the new server. The return value of the call in this case is the value of the previous setting. This allows one to temporarily substitute the default naming server with another and swap it back at the end of a collection of tasks.

At this point, as a client, we can refer to arbitrary servers. Now, we look at how we can use the services of these servers.

3 Identifying & Invoking Methods

Unlike $\widehat{\Omega}$, we cannot make calls to CORBA servers completely transparent. Instead, we use an interface similar to the $.C()$ and it is called $.Corba()$. This takes the object identifier - a character vector specifying the IOR or an name in the naming service. The second argument is the name of the operation to invoke. After these two required arguments, one passes the arguments for the operation.

After this, other arguments are passed to the CORBA operation. Usually, the arguments are passed as simple values and in the same order as defined in the IDL operation. However, one can also provide named arguments, using the names of the IDL parameters.

Two other arguments can be explicitly supplied to control more precisely how the operation is invoked. These are discussed later on in text.

Let us suppose that there is a CORBA object server that implements the *Matrix* defined above and that it has been registered with the naming service under the name *mat*. (We will see how to do this in later sections.) Then, we can find out the number of rows and columns it has with the calls

```
> .Corba("mat", "nrow")
[1] 3
> .Corba("mat", "ncol")
[1] 4
```

Now that we know the dimensions of the matrix, we can extract entries. Because the server was implemented to use 0-based indexing, we get the element at position (i, j) , such as $(2, 3)$ with a call

```
> .Corba("mat", "data", 2-1, 3-1) # or .Corba("mat", "data", 1, 2)
[1] 2.34
```

We can implement a version of a *Trace()* function as

```
Trace <- function(m) {
  r <- .Corba(m, "nrow")
  c <- .Corba(m, "ncol")
  total <- 0.
  for(i in 1:min(r,c)) {
    total <- total + .Corba(m, "data", r-1, c-1)
  }

  total
}
```

In S4, we can make the $.Corba()$ calls reasonably transparent with a little work.

```
setMethod("nrow", signature("CORBAObject"), function(x){.Corba(x, "nrow")})
```

Then, we can invoke this method as

```
nrow("mat")
```

Unfortunately, this approach requires that we create methods for all possible operations. This can be done programmatically by interrogating the Interface Repository for its IDL definitions and then set the methods automatically. See the functions described below for querying the naming service.

In $\widehat{\Omega}$, we can refer to the CORBA objects as if they were local variables and then invoke methods directly. For example,

```
x = mat.nrow()
```

At this point, we should discuss how the arguments to the IDL operations are handled and return values are converted to regular objects in the mechanism underlying the `.Corba()` function. We present this in some detail so that users may understand their responsibilities.

Basically, CORBA does not understand arguments in the form we specify them since they have their own representations. However, once we identify the server and the method name, we can obtain the signature of the IDL operation. At this point we know the expected type of each parameter and we can attempt to convert the user-level argument to the appropriate CORBA type. The basic IDL primitive types - long, string, char, double, float, boolean, etc. - are automatically handled by explicit code provided for each system supported by this distribution. Similarly, sequence of these primitive types are handled to convert between vectors in the system and CORBA.

The *Matrix* uses just these primitives as arguments and return values - long and double. However, clearly it is not convenient to specify parameters only in terms of these types. This would lead to operations with enormous numbers of parameters. Instead, we want to deal with higher level objects which are implemented using primitives. For example, we may want to pass a matrix to a scatterplot server to produce all pairwise plots.

Much research and many implementations have dealt with this. One approach is to require the user to convert the data to a form understood by the remote server. For example, we might pass the matrix as an array of doubles with the first two values containing the number of rows and columns. Alternatively, we might use more system-specific format such as that used by the `dump()` and `restore()` functions in S. This approach was used in an S-JavaTM interface [?]. Another alternative is to talk in the language of the remote interpreted server. If this was an S server we might call

```
eval("plot(matrix(c(100,4, 1.2,.....)))")
```

And we would have to handle the return type in a similar manner. That is we would get a string that we evaluate to produce a user-level object. This is very inefficient and we lose all the benefits of type checking provided by IDL. If we were to replace this server with one implemented in another system, e.g. Matlab, we would have to change the client implementation to produce and understand different strings.

A more useful approach is to convert non-primitives into lists of primitives and to operate recursively. In this way, a structure containing a long value and another list containing a string and a double would be mapped to the corresponding list-type in the target system and the regular primitive converters applied to the different elements.

John Chamber's S-JavaTM interface also distributed by the Ω project does the conversions between the two representations in this way. This is convenient for the user who does not want to deal with providing converters. However, it can be quite expensive in that all data is copied just to compute the number of rows and potentially determine that the object is not invertible, etc. Additionally, the actual resulting representation in the target system may need to be converted again for use by functions, methods, etc.

An alternative to all of these approaches and one that requires the least amount converter code and unnecessary data transfer (at the expense of increased number of transfers) is to leave the non-primitive user data used as argument to a CORBA operation in its own environment. Then we arrange to have methods invoked on it by the remote object (original server) to which it is passed. For example, an object implementing the *Matrix* interface might be passed to a server that will compute its trace. That function or method would then be implemented by invoking the `nrow`, `ncol` and `data` methods on that argument. The `Trace()` method above is an example of this that we will return to later.

One important consequence of this approach is that, as per the notion of IDL, the implementation and representation of a CORBA object is completely encapsulated. Other objects can only access it in terms of its IDL operations rather than assuming a particular structure. This allows us to represent the data in interesting ways appropriate for the resources of the machine on which it resides, etc. This in turn allows us to develop a different implementation of the same IDL interface which may be faster or more efficient with memory use, for example. Or perhaps we may have an optimizer with better properties but accessible as the same IDL interface. This allows us to easily substitute one object with another implementing the same IDL interface. This modularity is important for extensible and maintainable software. *This approach is the essence of CORBA.*

Note also that this approach allows a call to an operation in a server A to pass a reference to a CORBA object residing in a third process without transferring the data twice. `.Corba("plotServer", "pairs",`

So now we have discussed what we want to have happen - creating a CORBA object from a local object implementing a specified IDL interface and then to implement the methods of that interface locally using that object's data. CORBA allows us to do this by

- compiling the IDL skeletons that *marshall* the data¹,
- implementing the methods of the interface,
- linking these into the interpreted environment, and
- providing wrapper functions to interface between the *C/C++* routines and the interpreted language.

Since this involves much work and a certain level of expertise, we provide a simpler solution.

The Dynamic Skeleton Interface CORBA provides allows us to embed the user-level object in a proxy or virtual CORBA object that claims to implement a particular IDL operation. When this virtual CORBA object receives a request it extracts the arguments and converts them to local user-level objects. In the packages these and the name of the operation and the user-level object being represented by this virtual server and passes it to a user-level function. At this point, the appropriate interpreted function is called. In the case of the *nrow* operation of the *Matrix*

When the *.Corba()* attempts to convert a non-primitive

Many operations involve just **in** or **inout** arguments. However, there are some that take **out** arguments. For these, a value NULL suffices, however it is unnecessary to pass a user-level argument as an argument. However, for those If any of these are named, then all must be named (in the current implementation). The names must match the While there are two other prescribed arguments,

Non-Primitive Arguments lead to CORBA Servers.

4 CORBA Servers

We have seen how we can invoke methods in servers outside of the user's environment. We have also seen how the *.Corba()* implicitly creates CORBA servers when it encounters a non-primitive parameter type. A natural extension of this is for a user to create their own CORBA servers from within their environment. We can do this in the same way that arguments are converted to CORBA objects - via a proxy or virtual CORBA server. The requests are dispatched to interpreted functions (which of course may call native code, other CORBA methods, etc. in their implementations).

In order to create a server implementing a particular IDL interface, one first must define the functions to handle the requests. The simplest way to do this is to define them in the current working database with the same names and arguments as the IDL expects. Of course, for languages like *S*, *R*, *Matlab*, etc. there is no implicit default object on which a method is invoked. In other words there is no *this* or *self*. Thus, to obtain the user-level data contained in the CORBA server that provides the data to parameterizes these methods (i.e. *nrow*) these functions must accept an additional argument to those specified in the IDL. By convention, this will be passed as the first argument, before the CORBA operation arguments.²

The function that explicitly creates a CORBA server is *.CorbaServer()*. It takes as its first argument the name of the IDL. This is assumed to be in the Interface Repository and by default, this is checked when the server is instantiated.³ The following argument(s) is a user-level object(s) to be used as data for the server in the same way that an argument value is embedded in the virtual CORBA object and passed to the user-level implementation functions.

```
.CorbaServer("StatMatrix::Matrix", matrix(rnorm(100),10,10))
```

Note that we can use either *StatMatrix::Matrix* or the more complete *IDL:StatMatrix/Matrix:1.0* form for the IDL type.

A server is of little value if no other application can locate it and access its operations. Thus we need a mechanism of registering it with the naming service or obtaining its IOR. A simple way to do the former in languages like *S* and *R* is to use argument names as names in the top-level interface being implemented naming context to which the object is bound. For example, the call

¹Marshalling is the communication of the arguments between process and machines. This takes care of handling failures in network connectivity, converting data between machine representations, etc.

²The functions can also take other default arguments or other constructs that the language supports. One can also change the order that the arguments are passed.

³This can be suppressed via the argument *checkTypes* to allow a server to be created before its IDL interface is added to the repository. This is usually a logical value indicating whether to check or not. If this value is **T**, if any IDL type is not located in the repository an error is raised. However, if this is an integer value different from 0 and 1, warnings rather than errors are raised. This notifies the user of what IDL types must be added with out being a fatal error.

```
.CorbaServer("StatMatrix::Matrix", mat = X)
```

creates a CORBA server that implements the *Matrix* interace using the object *X* and the user-level functions and makes it accessible to other CORBA objects under the name *mat*.

If we want to register the object in a sub-naming context, we cannot use this mechanism. *S* for example does not support the syntax

```
foo(c("A", "B") = X)
```

So, instead we use the *name* argument of *.CorbaServer()*. This is can be passed a character vector specifying the nested name as described in section ???. For example, to create the server with the name *mat* in the naming context *Duncan*

```
.CorbaServer("StatMatrix::Matrix", mat = X, name=c("Duncan", "mat"))
```

Note that sometimes we create a server that provides functionality but that does not require any auxiliary data. Specifically, there is no equivalent of *this*. In these cases, one can pass the `NULL` value as the user-level interface being implemented object for the server. In this case, this will not be passed as the first argument to the functions called when implementing the IDL operations. This allows us to avoid having to create simple wrapper functions to remove this argument.

This example leads us to consider cases where we may want to map operation names to functions that do not have the same name as the operation. For example, in the *Matrix* interface above, the *data* corresponds the *l()* in *S*⁴. In this case, we may want to provide a different dispatching function that recognizes the operation name *data* as a special case. We can define this function and assign it to a name, say *MatrixServerEval()*.⁵ The newly created server can be instructed to call this with the CORBA operation parameters and other data by specifying its name as the value of the *evalFunc* argument. This allows us to map requests to arbitrary functions or expressions and basically do method dispatching based on all the elements of the request, including the return type.

4.1 Mutable Servers

Servers, by definition, provide services to callers and these come in the form of operations or methods. Some of these methods may change the contents of the server. For example, the *setElement* operation in the *BaseMatrix* interface is designed to change the value of an element of the matrix. Both *S* and *R* are not designed to support the notion of methods within an object that can change the value. Instead, these are functional languages that return copies of altered data, rather than modifying the single instance of the object. This poses a problem for us in the CORBA world where the semantics of method calls are very different. Fortunately, we can solve this difficulty and at the same time circumvent the problems of conflicting names and the implicit *this* reference in the IDL operations. Unfortunately, the mechanisms employed in *S* and *R* to handle these problems are different. We will discuss the *R* structure first.

R provides the concept of a closure which comes very close to offering *JavaTM*-like classes. They offer the nested name space of methods and an almos transparent *this* reference and mutability. Just as we would in *JavaTM* or *C++*, we collect the methods for an CORBA server into a closure and then create an instance of that closure. For example, consider the *BaseMatrix* class.

```
baseMatrixGenerator <- function(this) {
  nrow <- function() {
    return( super(nrow)(this))
  }

  return(list(nrow=nrow))
}
```

6

⁴This is true if we assume that the arguments are in terms of a 1-based indexing system.

⁵We will discuss how to implement this in section ??.

⁶The *super()* function retrieves the specified function by ignoring the ones defined locally within this closure and looking in the search path.

```
super <- function(fun) fun <- substitute(fun) if(is.name(fun)) fun <-
```

4.2 Multiple Servers

The `.CorbaServer()` as described above allows us to create a single server within the user-level environment. The CORBA mechanism however allows us to support multiple servers within an application simultaneously. This allows us to have just one process but many servers all available to CORBA clients. Since *S* and *R* allow only a single evaluator, we must ensure that requests to the dynamic servers are queued and executed one at a time. Fortunately, most CORBA implementations help us here by offering different *concurrency* models. Typically, we can arrange to have multiple servers residing in the process with just a single listener for all requests. This listener then hands the request to the appropriate COBRA server and does not execute the next request until control is returned to it by the CORBA server. This is exactly what is needed for these non-threaded systems. It is this that allows us to make a call as a client with CORBA objects as argument to the operation and block waiting for the result but still allow the original server to invoke methods in the argument objects that are executed within our local environment. These are termed nested calls or callbacks and understanding this was the reason for the lengthy delay in the release of this software.

For more advanced systems that support threading (e.g. Ω and native applications), there are other concurrency models we can employ. We can create a thread for each request or allocate a thread from a fixed pool for each request. These allow multiple requests to be executing simultaneously, potentially for the same server. If this is not desirable, we can direct the central listenern to serialize requests for each server but to allow requests for different servers to be handled concurrently.

So now that we know we can make better use of our environment and offer multiple servers simultaneously with little cost to our programming effort, we need to know how to do this. There are two basic mechansims. The simplest exploits the fact that the function `.CorbaServer()` is vectorized in several of its arguments. By this I mean that if you specify, for example, more than one IDL type as the first argument in the form of a character vector, it will create that many servers. In the traditional *S* manner, it will cyclically replicate the other arguments and create that many servers.

The following call creates three servers

```
.CorbaServer(rep("StatMatrix::Matrix", 3), m1)
```

each implementing the IDL interface `StatMatrix:Matrix` and each parameterized by the value of *m1*.

Since no names were specified, the IOR's are displayed and the terminal and can be manually transferred to other applications so that they can access these objects. Alternatively, we can specify names for the servers via the *name* argument. This can be done as

```
.CorbaServer("StatMatrix::Matrix", m1,
             name=list("A",c("Folder", "B"), "C"))
```

Note that we did not replicate the IDL types but instead defined the number of servers by the number of names, the length of the argument *name*. Note also that this argument is a *list* rather than a character vector. This allows us to describe nested CORBA names, as the second element does. In this case the first two arguments - IDL type and user object - are replicated.

Similarly, we can specify multiple user-level objects. The call

```
.CorbaServer("StatMatrix::Matrix", A = m1, B = m2)
```

creates two servers from the objects *m1* and *m2* and names them A and B.

In al of the examples above, all the servers are created and then the process enters the request listener loop. It only returns from this when we interrupt it (with a signal - Ctrl-C) or when all of the servers have been deactivated by external requests. This is useful if we can create all the servers in a single call. However, sometimes this is not convenient or even possible. Instead, we may need to create one group of servers and then a second in different calls. This may be the case for example when iterating over a structure to create servers from the different elements. A simple call to `lapply()` or `for()` are examples of this scenario. Consider what happens in the following expression:

```
for(i in 1:length(dataList))
  .CorbaServer(type{\Tt{i}}, dataList{\Tt{i}}, name=names(dataList)[i])


---


as.character(fun) fun
get(fun, env = .GlobalEnv)
```

The first server is created and the call to `.CorbaServer()` does not return until the server is deactivated. This is not what we intended. Instead, we wanted all the servers to be created and then to enter the request loop.

The argument `block` of the function `.CorbaServer()` is designed to support the need to create multiple servers in different calls. By default its value is `T` and hence the call enters the request loop and does not return until it exits that loop. However, if the value `F` is specified, the `C` code does not enter the request loop but simply creates the server, registers it with the naming service if appropriate and returns. These are what we term *deferred* servers. They are real CORBA objects at this point but not active. Any requests dispatched to the object will fail because there is no listener for these requests.

Using this argument, we can reimplement the loop above now as

```
for(i in 1:length(dataList))
  .CorbaServer(type{\Tt{i}}, dataList{\Tt{i}}, name=names(dataList)[i],
              block= (i==length(dataList)))
```

This simply instructs the `.CorbaServer()` to block only when executing the last command. Of course, if we had two of these loops, the first would pass `F` as the value for `block` for all iterations.

When the value `F` is passed for `block`, the return value of the call to `.CorbaServer()` is the CORBA object. We should also that the user-level object used by the server is copied to an appropriate storage frame/area so that it is available when requests are executed. This is necessary in the deferred mechanism.

5 Deferred Tasks

One of the two primary motivations for the CORBA interfaces to the different systems discussed here was to allow distributed high performance computing. A simple and canonical example of this that we dispatch each iteration of a loop to a different server to perform a sub-task of the overall task and that these are done concurrently. The obvious result is that the overall time to do the complete task is approximately divided by the number of sub-tasks we use. Again, consider a implementation of cross validation using this distributed setup. We assume that the `dataList` contains suitable parameterization of the data⁷ so that server receiving that element can determine what to do.

```
n <- length(dataList)
ans <- vector("list", n)
for(i in 1:n) {
  ans{\Tt{i}} <- .Corba(server{\Tt{i}}, "crossValidate", i, dataList{\Tt{i}})
}
```

Again, just as with the creation of multiple servers, the first of these calls will block until that server responds with the result.⁸ Then, the second sub-task is performed. While this may still be better than doing the computations locally (because the other servers may be faster, etc. and there is no need for synchronization) in many cases it would be preferable to have them perform the sub-tasks.

What is needed in the above case is the ability to not wait for the result of an operation in a remote server but to be able to do something else while it is executing and to return when it is completed. This is what is termed a *deferred* call in the CORBA literature. There are two basic ways to use this in our environment and while they differ conceptually, they are specified simply by changing arguments to the `.Corba()` call.

The first deferred call mechanism involves dispatching each request separately but specifying a value for the argument *deferred*⁹

⁷This is highly non-trivial as different configurations - distributed data, shared data via Network file systems, databases, etc. - will give very different characteristics for different classes of problems.

⁸Alternatively, an error may be returned in the form of an exception and then that sub-task must be rescheduled.

⁹You may wonder why such odd names are used for the arguments in the `.Corba` call. It took a bizarre occurrence for me to figure out what is subtly obvious. The arguments expected by the IDL operation are specified in the `...` argument of `.Corba()`. As explained when discussing out and inout arguments, the user can specify the IDL parameter names in these lists to identify the arguments. However, the `...` mechanism in Scauses all the arguments to be matched immediately and the remaining unnamed ones to be matched in a slightly different way. So in one example, I had an IDL parameter named `'1'` and the second argument to `.Corba()` was named `methodName`. Then in the call `.Corba("foo", "trace", m=m1)` the partial matching identified `m1` as the value for `methodName`. Thus to ensure that we avoid any conflict with arguments to this function and IDL parameter names, we prefix the local argument names with ``` which is not legal in IDL(?). Thus, there is possibility of matching `m` to `.methodName`

5.1 Specialized Converters

When an user-level object is converted to a non-primitive CORBA object, the default converter creates a proxy object that contains the user-level object and implements the CORBA methods by invoking functions of the same names as the operations. This is convenient but we may want to override this behaviour in at least two circumstances. The first of these is when we want to use a different mapping between the CORBA operation names and the user-level functions that get called. The second is when we want to avoid the entire interpreted mechanism, implementing operations with user-level function calls. Instead, we may want to implement the server's operations directly in the native language - e.g C. Both of these situations are handled by registering a new conversion rule using via the function `addCorbaConverter()`.

In the first case, we want to replace the default user-level function that acts as the bridge between the CORBA operation and its evaluation via user-level functions. This is `.CorbaServerEval()`. We can arrange to have a different function, say `foo()`, called for all requests for a CORBA object of type `bar` with the call

```
addCorbaConverter("HandlerForFoo", "IDL:foo:1.0", userFunction=T)
```

The same function can be used to register a C routine to be used to create the CORBA object. This is for more advanced users and is primarily used for efficiency purposes. Avoiding the calls to user-level functions can lead to huge performance gains at the expense of flexibility. How this object implements the CORBA method requests is its responsibility. The C routine must have a signature

```
CORBAAny_ptr (*converter)(CORBAAny_ptr target, const USER_OBJECT_ &userObject);
```

The second argument is the user-level object which we want to convert to a CORBA object. The first argument is the container into which the new CORBA object is to be inserted for communication with other routines. Generally, one creates a new CORBA object (via one of its constructors) and then inserts it into the target container:

```
*any << = newCorbaObject ;
```

(We do not do this automatically since some users will need to create their own Any object. In this case, this new Any should be returned.)

The return value is almost always be the first argument. Only in very rare cases will one create a new Any into which the newly created CORBA object is added.

To register such a function, we specify its name (as it would be named in a call to `.C()`) and the target IDL type.

```
addCorbaConverter("routine-name", "IDL:foo:1.0")
```

Note that we currently do not provide the resolution to specify the converter for a specific object or for a particular user-level class.

In the future, if it is deemed useful we may provide a "method" table mapping user-level classes to particular routines. Thus, for a given IDL type, we would be able to specify different routines or user-level functions for different user-level types/classes. For the IDL type, this would be a table of the form

list	"CORBAfromList"
matrix	""
myClass	

Note that inheritance is not used in the lookup of converters. Specifically, if we converting a user object to a CORBA type, say `bar` which extends `foo`, then we only look for a converter registered for `bar`. We do not look for converters for its IDL base classes, and theirs, etc.

This maps an IDL type to either a user-level function that replaces `.CorbaServerEval()` in providing the bridge between the CORBA proxy server and the interpreter

While typically one would register a

This can be done by supplying

```
addCorbaConverter()
```

6 High Performance Computing - Distributing Tasks

What we have discussed above is the ability to invoke methods in remote objects in arbitrary languages and create such remote servers using the existing high level languages only. This allows us to share functionality with other

applications. A second usage is to be able improve the performance of existing algorithms within a language in a simple and straightforward manner by using more resources to in the computation of the algorithm. Given the ability to communicate with objects in other applications, we can now decompose a computational task, say T , into sub-tasks T_1, \dots, T_k . Generally, the overall task is completed when all of the subtasks are completed. However, in many cases the different subtasks can be done in any order. This is especially true for many statistical applications such as bootstrapping, cross-validation, certain fitting procedures (e.g. nlme for within-subject estimation). While a decomposition of the task into subtasks can be challenging, once a suitable one is derived, the sub-tasks can be performed on different machines. In an ideal setup where there are k machines, 1 for each sub-task, the overall computation time for Y can be reduced by a factor k . While this is rarely realized, there is usually significant gain. There are however many issues such as configuration of the slave machines, initialization of the servers on these machines, transmission of the data, etc. Clearly the improvement in performance is related to the complexity of the sub-tasks. If each of these is trivial, the expense of the remote method invocation will outweigh potential gain from distribution. However, if this is a negligible component of the overall time for task T_i , the gains can be significant.

We consider a simple example here to illustrate how one might use this high-performance cluster of servers from a language such as S . In this case, the tasks are homogeneous and involve cross-validation on a model. The auxiliary data common to all the sub-tasks have been pre-allocated in the remote servers. This includes the previously fitted model, the data, etc. Each sub-task T_i involves recomputing the fit with a certain sub-group of the data being removed and then predicting the value. The result is a single real value, the prediction error for the omitted group based on the fit from removing this group from the initial data. The inputs to the sub-task (in addition to the fixed fitted model and data) are simply the indices of the elements of the sub-group. Again, for simplicity, we will assume we 10,000 observations in the data and each sub-group and hence sub-task works on 1,000 of these. Hence the inputs for sub-task i are the indices

$$\pi(j) | 1000 * (i - 1) + 1 < j < 1000 * i$$

where π is a permutation of the values $1, \dots, 10000$.

A simple example of this The following

7 Event Channels

8 The Interface Repository

Reflectance is the facility by which an object programmatically provides information about its makeup. For example, in $Java^M$, we can interrogate a class for the different methods it supports, the fields an instance contains, and we can even create objects of this class by calling different constructors, invoke methods and access these fields without compiling the calls directly. This might be defined as meta-programming and it lies at the heart of the $\hat{\Omega}$ interpreter. S objects also support reflectance - what is the definition of the class, its superclasses, the methods defined for it, etc.

The dynamic mechanism for CORBA described here also uses reflectance. This comes from the Interface repository which allows us to explore an IDL module or interface and its sub-elements. Since we have been able to exploit it to implement powerful facilities simply, it seems that we should also provide user-level/interpreted access to these same facilities.

The function *idlType()* returns a string identifying the IDL type of an object. The object is specified as a name or IOR. A more interesting function which takes either a name/IOR or the IDL type is *idlInterface()*. This returns a list describing the IDL interface associated with the argument. Generally this will have 4 elements:

- a list containing the operations descriptions
- a list containing the attribute descriptions
- a character vector containing the names of the parent IDL interfaces from which this one was derived or extended.

- the full name of the IDL interface

The third value allows us to work recursively to discover all the methods and attributes accessible within an IDL type that are inherited from ancestor classes.

The function *getIRContainerContents()* returns a simple description of the contents of a repository object. Each element of the resulting list contains the identifier for the IDL element and its type (e.g. attribute, interface, module, etc.) This can be used to construct most of the information that is needed programmatically to identify the characteristics of CORBA objects. It works not just on repositories, but arbitrary containers, including interfaces, etc. It offers a condensed version of the output from *idlInterface()* that can be used when not all the information about operations and attributes is required but just the names. (Operations, attributes, and some other types are not containers so we cannot examine them in this manner.)

This querying of the repository may be extended to support other IDL elements such as a module, exception, struct, etc. This is not vital at present since $\hat{\Omega}$ offers both a graphical and programming tool to view (and edit) the Interface repository.

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

Additionally, from within Java and directly within $\hat{\Omega}$, one can add attributes and operations to an IDL interface and even create new IDL elements without creating an external source file. The elements are added directly to an interface repository.

Like the Naming server, there can be more than one interface repository and each is a CORBA server. We can change the identity of the default repository using the function *interfaceRepository()* as well as simply obtaining a reference to it.

One of the drawbacks of the dynamic mechanism is that we need to populate the Interface repository with the interfaces we are interested in and before we employ elements of these interfaces. The compiled versions do not require this. For this and efficiency reasons, it is often a good idea to compile stable IDL stubs (not necessarily skeletons) for use with clients. Now if we make the *.Corba()* mechanism aware of this compiled class the local system (e.g. *S*) can also create appropriate *.C* calls to the different operations and narrow the object using the helper classes.

8.1 Named Objects as Variables

$\hat{\Omega}$ allows a user to refer to objects in a CORBA naming context directly as if they were regular local $\hat{\Omega}$ variables. Additionally, methods can be dispatched as if they were local as the equivalent of the *.Corba()* call is done implicitly if there is no local method matching the one being invoked. For example, in the examples above, we can write

```
mat1.nrow()
```

having “attached” the toplevel naming context as an element in the evaluator’s search path.

S4 allows us almost allows us to attach naming contexts as databases in the search path. For certain reasons, a decision was made to prohibit databases that can be modified externally to be used as elements of the search path. Instead, objects contained within them can only be referenced explicitly via *get()*. The machinery is there however to fit this framework as long as we do not want to by-pass the internal lookup table associated with an attached object. So, in what follows, we consider the contents of the naming service static.

The classes *CORBANamingContext* and supporting *CORBAObject* and *CORBAName* and their associated methods for *dbobjects()*, *dbexists()*, etc. provide the basic facilities. What we would like to be able to do is something like the following:

```
> attach(CORBANamingContext())
> .Corba(Server, "nrow"
```

where ‘Server’ is now retrieved from the naming context and is passed to the *.Corba* as an object of class *CORBAObject*. Note that this is slightly more expensive than specifying the string to *.Corba()*. It becomes more useful when we have multi-level nested names. There we can attach the sub-naming context directly to make the objects contained within it directly available as top-level variables.

```

> attach(CORBANamingContext(), name="Default Naming Service")
> objects(2)
 [1] "A"           "B"           "BAR"         "C"
 [5] "D"           "Duncan"     "E"           "ExtendedServer"
 [9] "F"           "FOO"        "OP"          "PO"
[13] "Sarea"      "Server"     "The"         "buf0"
[17] "foo"        "smat"      "smat1"
> smat1
[1] "IOR:010000001d00000049444c3a6f6d672e6f72672f434f5242412f4f626a6563743a312e30000000"
>

```

Similarly we can attach sub-naming contexts as

```

> attach(CORBANamingContext("Duncan", "Temple"), name="Duncan")
> objects(2)
[1] "Lang"

```

Alternatively, using the syntax developed below

```

> attach(CORBANamingContext(Duncan$Temple), name="Temple")

```

We can take this one step further and define a method for the \$ operator for CORBA objects. We can attempt to narrow these to naming services and lookup the sub-element within it. For example,

```
Sarea$matrixServer
```

would mean lookup the object *Sarea* and then invoke the method \$ with the argument *matrixServer*. This would then extract the named object *matrixServer* within the naming service. This now works correctly.

In the case that source object (i.e. the lhs of the \$ operator) is not a naming context, then we attempt to extract the attribute corresponding to the name on the rhs of the operation. For example, in our *ExtendedTraceServer* IDL example, each server has a variable **count**. So

```

> ExtendedServer$count
[1] 13
>

```

should return the current value of that object.

At present, assigning a value in this manner requires that the object be local to the default database.

```

> ExtendedServer$count <- 100
Problem: object "ExtendedServer" must be assigned locally before replacement
> v <- ExtendedServer
> v$count <- 100

```

Unlike $\hat{\Omega}$, the syntax of *S*, *R*, etc. does not support hiding the method dispatching in a general and centralized way. We can however arrange to create methods for the *CORBAObject* and derived classes for certain methods. For example,

```
setMethod("nrow", "CORBAObject", function(x){.Corba("nrow", x)})
```

or

```
setMethod("nrow", "CORBAMatrix", function(x){.Corba("nrow", x)})
```

Using the facilities in the earlier section (??) we can enumerate the interfaces and their operations within the interface repository and automatically generate these methods.

Another “hack” that we can implement to provide syntactic sugar for CORBA calls is to overload the \$ operator one more time. Here we can have

```
> ExtendedServer$random(10)
[1] 0.63397012 0.05028742 0.02118261 0.57749794 0.73148933 0.06757584
[7] 0.41231109 0.63403648 0.15342002 0.14735986
```

invoke the IDL operation *random* of the object *ExtendedServer*.

How is this done? It relies on knowing the representation of an *S* function and also the evaluation model for an expression of the form

$$a\$b(x, y, z)$$

Such an expression is evaluated, as one might expect, as a call to the object returned from evaluating $a\$b$ and calling it. In the case of a being a CORBA object and b being an attribute of the IDL, we could evaluate it call there. In the case of a function call, we dynamically create a function to return as the value of $a\$b$ which substitutes the values of the CORBA object a and the operation name b into the function that calls *.Corba()*.

The entire method for the $\$$ operator then is made up of the following sub-sections.

- a is a Naming Service Context
- b is an attribute in the IDL interface of a
- b is an operation in the IDL interface of a

13 \langle GPL License 13 $\rangle \equiv$
Copyright (c) 1998, 1999 The Omega Project for Statistical Computing.
All rights reserved.

\langle GPL License 13 \rangle