

This is the no-frills short manual or guide to some of the CORBA accessor functions in *S* and *R*. We illustrate the functionality using basic examples. We are not concerned with motivation here.

The simple example we will use is to put a user-level matrix on the CORBA bus and have another session/process communicate with it and invoke methods it supports. First, we need some IDL. We define a very basic interface for the IDL Matrix which we define in an IDL module named StatMatrix.

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

Notice that we use ‘;’ to end each enclosing block/brace. The module can contain multiple interfaces, constants, etc.

The operations we define provide enough to access the data and reconstruct the matrix elsewhere. They are sufficient for doing most computations, if inefficiently. Later, we can extend this class.

The IDL above has defined the contract between the server and client. Next we must arrange to install the basic services the CORBA uses (in our setup). These are the naming service and the interface repository. For Orbacus, these are instantiated with the commands **irserv** and **nameserv**. We must ensure that they are running in a well defined and easy to find location known to the clients. Setting the value of the environment variable ORBACUS_CONFIG to the name of a file allows the different Orbacus utilities to rendezvous on these known service points. Such a file exists in **OMEGA_HOME/Interfaces/CORBA/CORBAConfig/Orbacus.config** and basically contains machine-port pairs for the different services.

```
setenv ORBACUS_CONFIG $OMEGA_HOME/Interfaces/CORBA/CORBAConfig/Orbacus.config
irserv &
nameserv &
```

Note that ports these listen on are specified by us and are not requirements. We use the same ones so that we can put them in a configuration file.

Since we are going to be working in a dynamic world rather than compiled CORBA environment, we must make the IDL defined above available to servers and clients. We do this using **irfeed**.

```
irfeed StatMatrix.idl
```

Before we do anything concrete with CORBA in a session, we must arrange to initialize the connection to the ORB. This can be done using the *CORBAinit()* function. It is convenient to call this in a *.First()* for the chapter. This function allows us to pass “command line” configuration arguments for the ORB and BOA. The easiest thing to do is to have a configuration file for the CORBA implementation which specifies the “location” (or IOR) of the naming service and interface repository, and any other services.

Now, we can consider creating the server. We use the *.CorbaServer()* to create an object and we make it accessible to other CORBA objects by adding it to the naming service with the name `MatrixServer1`.

```
.CorbaServer("IDL:StatMatrix/Matrix:1.0", MatrixServer1=matrix(rnorm(100),10,10))
```

This indicates that the newly created matrix object should be used to implement the IDL interface *IDL:StatMatrix/Matrix:1.0* and to bind it to the name `MatrixServer1`.

Of course, we could have overwritten an existing server with the same name. If this was not our intention, we need a mechanism to examine the currently used names. The function *namingServiceContents()* returns a list of the named elements in the top-level naming context. Each name has an indicator as to whether it is a sub-naming context or a regular object. A simpler graphical interface is available from $\hat{\Omega}$.

The other arguments to *.CorbaServer()* allow remote access to the object to be deferred (the *block* argument); nested names to be used to locate the object within sub-naming contexts. A special argument is *evalFunc*. This is used to handle CORBA requests and dispatch them correctly. By default, this is *.CorbaMethodEval()* which looks for a function of the same name as the IDL operation name and passes the CORBA arguments to it. Different functions

can be specified here to dispatch these requests differently. For example, we may provide a function that maps certain operation names to different function names and then dispatches.

At this point, the object is on the CORBA bus and the session is blocked. There is no way to run this in the background since *S* and cousins are single-threaded.

The first two operations defined in the IDL will work correctly since there are functions of the same name that take a matrix as their sole argument. `.CorbaMethodEval()` takes care of supplying the user-level object used to create the server as the first argument of these functions. There is no `data()` function however. So, before we create the server, we should have defined such a function. This is very simple as we take its IDL definition and define it to take an extra argument which is the user-level object itself. Also we must add 1 to each of the index arguments since we have implicitly assumed that we are using 0-based counting. So a suitable function is as follows

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

At this point, we can create a fully functional server. It will block the process, waiting for requests and dispatching them, one at a time.

Note that we do not have to use global functions to implement the different methods. One can specify a different dispatcher function or in *R*, a closure which contains all the methods locally. This is the preferred mechanism when the methods mutate the user-object that defines the servers internal data. See **Overview.nw**.

Now, we turn our attention to the client. We start a second process in another window (potentially on another machine) Again, we initialize the CORBA connection using `CORBAinit()`. Now, we invoke one of the IDL operations on the server. Akin to the `.C()`, we use the `.Corba()` specifying the name of the CORBA server, the operation name and a list of arguments.

```
.Corba("MatrixServer1", "nrow")
```

This returns an integer vector of length 1.

We can create servers that have names in subcontexts using the *name* argument of the `.CorbaServer()` function.

```
.CorbaServer(c("IDL:StatMatrix/ExtendedTraceServer:1.0"), NULL, name=c("Sarea", "POOH"))
.CorbaServer(c("IDL:StatMatrix/ExtendedTraceServer:1.0"), NULL, name=c("Sarea", "First"),
```

Note that this assumes the naming context already exists.

To access it, we specify the same name argument to `.Corba()`.

```
.Corba(c("Sarea", "POOH"), "fastTrace", matrix(1:9, 3, 3))
```

The function `.CorbaServer()` is vectorized (in way too many arguments). This allows multiple servers to be created in a single call. For example,

```
.CorbaServer(c("IDL:StatMatrix/ExtendedMatrix:1.0"), A=m1, B=m2)
```

creates two servers with different data but implementing the same IDL interface. The names of the data arguments are used to register the new CORBA objects in the naming service. If nested names are required, the *name* argument can be used. This is a list of the same length as the number of user-level objects being offered as CORBA servers. The elements of the list are character vectors defining the nested name.

We can also specify multiple IDL types. The command

```
.CorbaServer(c("IDL:StatMatrix/ExtendedMatrix:1.0", "IDL:StatMatrix/Matrix:1.0"),
             m1, name=list("FOO", "BAR"))
```

creates two servers (with the same names but implementing different interfaces).

Some CORBA operations have arguments that are used to return data in addition to the regular return value. These are declared as either inout or out arguments. The former type of argument allows the caller to specify information potentially controlling the call and the callee to return information within the same object. The latter type contain no useful information for the callee and are just used to return extra values. In order to make things simple and to avoid


```
CORBAAny_ptr (*f)(CORBAAny_ptr target, USER_OBJECT_ &, bool isDeferred);
```

(The `&` is a *C++* construction implying a reference to the object, not quite a pointer!) When converting from a user-level object to a CORBA object, this routine is expected to create a valid CORBA object and insert it into the *Any*. The expected CORBA type is available from the *Any*. See *RSCorbaConverter* and the examples in **Test/** for more information. Also, see **Overview.nw** for information on potential speedup.

If a user-level function is specified, this is passed to the regular dynamic CORBA server and is called when a request is received. Note that functions are supplied by name and not by value as in a other interfaces (i.e. for use with `call_S()`).

While converters will be typically registered for transforming user-level objects to CORBA objects. However, one can register converters working in the other direction - from CORBA to user-level objects. These are identified by specifying the value **F** for the argument *toCorba*

Note that converter tables are stored in *C* structures and disappear when the session terminates. The information can be stored in user-level lists and the internal tables recreated in a future session from these lists.

2 Servers

In order to test things, one needs a client and a server. A server can be created within an *S* or *R* process very easily by using the *.CorbaServer()* function. The following call creates a server from a local matrix and allows

```
.CorbaServer("IDL:Omegahat/Matrix:1.0", Rmatrix=matrix(1:9,3,3))
```

3 License

Copyright (c) 1998, 1999 The Omega Project for Statistical Computing. All rights reserved.