Using the R (D)COM Server

Troubleshooting

If anything goes wrong when calling `.Init()` of the COM object for the first time, retrieve error information using `.GetErrorText()`:

"installation problem: unable to load connector"
R proxy library could not be loaded. The cause of this common failure can be:

- the environment variable `%R_HOME%` points to a directory where **R** is not installed *and*
- the registry key corresponding to your **R** installation does not point to the installation folder of **R** *and*
- `rproxy.dll` and `R.dll` cannot be found in the `%PATH%`

In case you have downloaded precompiled binaries from CRAN and installed them, something went wrong there. Please try to reinstall **R** using the setup program. Download "dbgview" (see section "Resources") to find out what's going wrong.

If you have compiled **R** yourself, check `%R_HOME%\bin\rproxy.dll` and the registry key `HKEY_LOCAL_MACHINE\Software\R-core\R\InstallPath` if this one points to your **R** installation folder.

"installation problem: invalid connector library"
possibly damaged **R** proxy library. This version of the COM server requires **R** >= 2.2.
"installation problem: interpreter interface version mismatch"
wrong version of **R** proxy library. This version of the COM server requires **R** >= 2.2.
"installation problem: interpreter version mismatch"
wrong version of **R**. This version of the COM server requires **R** >= 2.2.

If a message box shows up on your call to .Init(), which tells you that the base library could not be found, you probably should set `R_HOME` to your installation folder. This should rarely be necessary.

If you cannot solve the installation problem by yourself, please try to find help on the **R** COM mailing list (see "Resources"). Subscribing to this list is recommended if you want to use/are using the **R** COM server (or any other part of **R**'s COM connectivity modules).

All functions will return an error code corresponding to the texts shown above. This error code is the return value of the functions when called e.g. from C or C++ or can be retrieved by checking `Err.Number` in VB or VBA. The following table will list the error codes (defines for C/C++ and numbers)

Define	Decimal	Hexadecimal
SCN_E_INVALIDARG	-2147221503	0x80040001
SCN_E_INVALIDFORMAT	-2147221502	0x80040002
SCN_E_NOTIMPL	-2147221501	0x80040003
SCN_E_UNKNOWN	-2147221500	0x80040004
SCN_E_INITIALIZED	-2147221499	0x80040005
SCN_E_NOTINITIALIZED	-2147221498	0x80040006
SCN_E_INVALIDSYMBOL	-2147221497	0x80040007
SCN_E_PARSE_INVALID	-2147221496	0x80040008
SCN_E_PARSE_INCOMPLETE	-2147221495	0x80040009

```
SCN_E_UNSUPPORTEDTYPE           -2147221494  0x8004000A
SCN_E_EVALUATE_STOP             -2147221493  0x8004000B
SCN_E_INVALIDINTERFACEVERSION   -2147221488  0x80040010
SCN_E_INVALIDINTERPRETERVERSION -2147221487  0x80040011
SCN_E_INTERFACENOTFOUND         -2147221486  0x80040012
SCN_E_LIBRARYNOTFOUND           -2147221485  0x80040013
SCN_E_INVALIDLIBRARY            -2147221484  0x80040014
SCN_E_INITIALIZATIONFAILED      -2147221483  0x80040015
SCN_E_INVALIDCONNECTORNAME      -2147221482  0x80040016
SCN_E_INVALIDINTERPRETERSTATE   -2147221481  0x80040017
SCN_E_FATALBACKEND              -2147221472  0x80040020
```

Using the COM server (Example code in Visual Basic)

1. get a server object

   ```
   dim x as StatConnector
   set x = new StatConnector
   ```

   and install an error handler

   ```
   on error goto error_handler
   ```

2. fire up **R**

   ```
   x.Init ("R")
   ```

3. use **R**

   ```
   x.SetSymbol ("symname",value)
   ```

   or

   ```
   y = x.GetSymbol ("symname")
   ```

   or

   ```
   y = x.Evaluate ("expression")
   ```

   or

   ```
   x.EvaluateNoReturn ("expression")
   ```

4. shut down **R**

   ```
   x.Close
   ```

5. error handler

   ```
   error_handler:
   MsgBox x.GetErrorText,"R Server Error"
   ```

   You can retrieve Information about the COM Server by calling `x.GetServerInformation` passing the requested information identifier (see enum `InformationType` in `StatConnectorSrv.idl`). Get information about the proxy dll calling `x.GetConnectorInformation` or the **R** interpreter itself with `x.GetInterpreterInformation`.

Using the COM server (Example code in Python)

To use COM with Python, the Python for Windows extensions (available from http://starship.python.net/crew/mhammond/) have to be installed.

Thanks to Dominic Barraclough (mailto:Dominic_Barraclough@urmc.rochester.edu) for the following piece of code:

After firing up the python interpreter, if one enters the following we can see that -python can talk to **R** and get stuff back!

```
>>> from win32com.client import Dispatch
>>> sc=Dispatch("StatConnectorSrv.StatConnector")
>>> sc.Init("R")
>>> print(sc.Evaluate("2+2"))
4.0 # COMMENT- R can do arithmetic and can tell python about it!
>>>
```

Using the COM server (Example for APL)

This section was contributed by Grant Kilvington (mailto:gkilvington@edsellkilvington.com.au):

```
I am using R(D)Com with APL.  I hope the following may be helpful.  The
example I am using is taken from the book "Data Analysis and Graphics using
R, An Example-based Approach" by John Maindonald and John Braun.

First I load an APL workspace which has some functions for communicating
with R(D)COM.  Because you will have difficulty with the APL font we use, I
will describe the functions used - I think the approach would be the same in
all languages.

Then I initialise R by running Rstart.  This creates an instance of
'StatConnectorSrv.StatConnector' and then I apply the method 'Xinit' to it.
R is now running as a server.

Next I load the library "DAAG" by executing RExec 'library(DAAG)'.

RExec is a function which uses 'EvaluateNoReturn' exp (when I supply no left
argument) or 'Evaluate' exp (when I do supply a left argument).  I could
have had two separate functions depending on whether or not I want a result
returned to my APL session.

Now I load the data set "roller" by running the command RExec
'data(roller)'.  I can see that the data is in the R Server by executing 1
RExec 'ls()' (Note the <dummy> left argument) and this returns "roller" to
my APL session.

To see the values in APL I run weight {is assigned} 1 RExec
'as.vector(roller$weight)' and if I examine weight (a variable now in my APL
session) I see the values:

1.9 3.1 3.3 4.8 5.3 6.1 6.4 7.6 9.8 12.4

Note the phrase {is assigned} refers to a left pointing arrow which is an
APL symbol.  From now on I will simple use the <- construct used in R.

Similarly depression <- 1 RExec 'as.vector(roller$depression)' creates
depression for me with the values:

2 1 5 5 20 20 23 10 30 25

Now we use R to do a simple regression of depression against weight:

RExec 'roller.lm <- lm(depression ~ weight, data=roller)'
```

```
1 REexec 'as.vector(roller.lm$residuals)' returns to APL the values
¯0.9796694804 ¯5.179764594 ¯1.71311378 ¯5.713232672 7.953394365 5.819997622
8.019973844 ¯8.18012127 5.953037689 ¯5.980501724
and
1 REexec 'as.vector(roller.lm$coefficients)' returns
¯2.087147783 2.666745928

Sorry about the "too much" precision.  Note also that in APL the high minus
is attached to the number: the - sign is an operator (negate).

Note that I can also retrieve matrix results:

      {transpose} 1 REexec 'as.matrix(roller)'
  1.9  2
  3.1  1
  3.3  5
  4.8  5
  5.3 20
  6.1 20
  6.4 23
  7.6 10
  9.8 30
 12.4 25

I can also create R objects and then retrieve them into APL:

REexec 'dframe<-data.frame(roller,fitted.value=predict(roller.lm),
residual=resid(roller.lm))'

      {transpose} 1 REexec 'as.matrix(dframe)'
  1.9  2  2.97966948  ¯0.9796694804
  3.1  1  6.179764594 ¯5.179764594
  3.3  5  6.71311378  ¯1.71311378
  4.8  5 10.71323267  ¯5.713232672
  5.3 20 12.04660564   7.953394365
  6.1 20 14.18000238   5.819997622
  6.4 23 14.98002616   8.019973844
  7.6 10 18.18012127  ¯8.18012127
  9.8 30 24.04696231   5.953037689
 12.4 25 30.98050172  ¯5.980501724

Or I can extract bits of the frame as vectors via

      1 REexec 'as.vector(dframe$fitted.value)'
2.97966948 6.179764594 6.71311378 10.71323267 12.04660564 14.18000238
14.98002616 18.18012127
      24.04696231 30.98050172
      1 REexec 'as.vector(dframe$residual)'
¯0.9796694804 ¯5.179764594 ¯1.71311378 ¯5.713232672 7.953394365 5.819997622
8.019973844 ¯8.18012127
      5.953037689 ¯5.980501724

If I run the regression in APL (using my home grown function):

t1 <- 0.99 Regression ('depression' 'weight') (depression,[1.5]weight) 1 1 ©
Plot
Regression of depression on weight

Source        Sums of Squares     df   Mean Squares
=================================================
Model                  657.97      1         657.97
Residual               362.93      8          45.37
-------------------------------------------------
Total                 1020.90      9         113.43

Number of Observations                        10
F-statistic F(1,8)                            14.50
p-value, Prob > F                              0.0052
```

```
Variation accounted for (R-square)           64.45
              Adjusted (R-square)           60.01
 Root MSE (residual std deviation)            6.74

Variable         Estimate  StdError      t(8)  Prob>|t|      Mean
------------------------------------------------------------------
depression                                                   14.10
------------------------------------------------------------------
weight            2.66675   0.70024     3.808    0.0052      6.07
Intercept          ¯2.087     4.754    ¯0.439    0.6723

Correlation Coefficient:  80.28

The variable t1 contains a list of various results, for example t1[2] is:

   2  1.9  2.97966948  ¯0.9796694804
   1  3.1  6.179764594 ¯5.179764594
   5  3.3  6.71311378  ¯1.71311378
   5  4.8 10.71323267  ¯5.713232672
  20  5.3 12.04660564   7.953394365
  20  6.1 14.18000238   5.819997622
  23  6.4 14.98002616   8.019973844
  10  7.6 18.18012127  ¯8.18012127
  30  9.8 24.04696231   5.953037689
  25 12.4 30.98050172  ¯5.980501724

which is the same as the frame created in R (with the first two columns
reversed in order).

I can also use the R plot routines via (for example):

RExec 'plot(roller.lm, which=1)'

which creates the graphic in the R Graphics device which I saved as a jpeg
and attached as Rwhich1.jpeg.

Similarly the normal Q-Q plot via:

RExec 'plot(roller.lm, which=2)' (see Rwhich2.jpeg also attached).

I used a low jpeg quality to keep the size down (50

There is still lots of things I haven't worked out how to do yet.  In
particular I haven't been able to work out how to get R to pop up a text
window so that I can see the output from (say) summary(roller.lm).  Any
clues.
```

Using the COM server (Example code in Perl)

This section was contributed by David Ovelleiro (mailto:dovelleiro@gmail.com):

```
I've been using the COM server to interface Perl applications with the R
package, and the results are excellent. In the page
"http://sunsite.univie.ac.at/rcom/"-> usage, are some examples of using
the COM server with Visual Basic or Phyton. Maybe you can add an example
of Perl used together with DCOM and R, and maybe encourage the use of
your excellent solution in the Perl world.
The use is so easy as:

use strict;
use Win32::OLE;
my $R =Win32::OLE->new('StatConnectorSrv.StatConnector');
my @cars;
$R->Init('R');
$R->EvaluateNoReturn ('plot(cars)');
$R->EvaluateNoReturn ('vec<-array(,dim=c(50,2))');
$R->EvaluateNoReturn ('for(i in 1:50){vec[i,1]<-cars[i,1]}');
$R->EvaluateNoReturn ('for(i in 1:50){vec[i,2]<-cars[i,2]}');
```

```
@cars=$R->GetSymbol ('vec');
my @cars_formatted=@{$cars[0]};
for (@cars_formatted){
   print ${$_}[0]."\t".${$_}[1]."\n";
   }
```

RServerManager: A Short Introduction

R (D)COM server provides a mechanism for standard applications like Microsoft Excel or custom applications written in any language serving as a COM client (e.g. Visual Basic, Perl) to use the **R** as a powerful computational engine and renderer for graphics and text output.

The current implementation of the **R** server package puts every single **R** interpreter used in a client application into a separate address space, allowing different code and data segments for multiple instances of the interpreter even in a single client instance.

On the other hand, using COM/DCOM to expose **R**'s functionality to client applications even makes it possible to share a single instance of an R interpreter between multiple client applications, both running on the same or even on different machines in the network. Sharing an interpreter instance also implies a shared data and code segment for **R**.

The implementation using COM takes care of synchronizing access to the interpreter, so only one client can use the server's functionality at the same time.

Using these COM/DCOM features to share a single interpreter requires some level of "intelligence" in the client applications and cooperation between these. One client has to create the interpreter instance and all clients have to gain access to this object in some way, by using some kind of data exchange.

This is the situation where a generic concept for managing and sharing interpreters is required: the **R** Server Manager.

[Package Index]