British Journal of Mathematics & Computer Science
4(15): 2188-2196, 2014

SCIENCEDOMAIN *international*
www.sciencedomain.org

RCaller: A Software Library for Calling R from Java

M. Hakan Satman[*1]

[1] *Istanbul University, Department of Econometrics, Beyazit, Istanbul, Turkey.*

Original Research Article

Received: 15 April 2014
Accepted: 29 May 2014
Published: 06 June 2014

Abstract

R is a programming language and environment which is mainly aimed to be used for statistical calculations and data analysis. Since a vast amount of human resource is consumed for its source code and packages; it is comprehensive, large in size and more common among others. Extending computer software with statistical calculation routines requires extra human resource, therefore, the need of wrapper libraries has been appeared. There are many software libraries that communicate R with other languages. Despite they render similar services, they have their own pros and cons. RCaller is a software library which comes into prominence with its simplicity. In this paper we introduce the library with some examples. We mention its abilities as well as its limitations. RCaller can be used in relatively small projects as it has painless start-up process and steep learning curve.

Keywords: R; Java; Statistical Software
2010 Mathematics Subject Classification: 62-04;00-68;68P05

1 Introduction

R [1] is an open source programming language and environment for statistical calculations and data analysis. Since *R* is open source, there are hundreds of packages developed by the contributors around the world. Source code of *R* is compiled for many platforms and this is an other reason for fast growing of user and developer community.

R has many interfaces to communicate with the code written in foreign languages such as *C*, *C++* [2], *Fortran*, *Java* [3] and *Python* [4], among others. Interoperability of the language not only makes use of older software libraries written in other languages but it gives rise of extending new software with capabilities of *R* [5].

Java [6] is a general purpose programming language which is aimed to run on a *Java Virtual Machine* (JVM) [7]. Since a *JVM* can be implemented in several kinds of platforms, *Java* is supposed to be a *write once & run anywhere* language. This characteristic of *Java* is the main reason of its wide

*Corresponding author: E-mail: mhsatman@istanbul.edu.tr

use in web, desktop and mobile platforms. The popularity of the language implicitly increased the need for statistical libraries. Despite there are open source mathematics, optimization and statistics software libraries for Java [8], none of them are as comprehensive and scalable as *R*. This is the basic motivation under developing *bridge* or *wrapper* code between *Java* and *R*.

In this paper, we introduce a new software library for calling *R* from *Java* and show its capabilities. In Section 2 we refer to recent works and make a comparison with our library. In Section 3 we give a quick reference and some illuminative examples. In Section 4, we compare time performances of libraries that are mentioned in paper. Finally, we conclude.

2 R and Java Interoperability

There are many options to provide a calling convention for *R* in *Java*. *Rserve* is developed for communicating *R* and other languages including *Java* [9]. *Rserve* is a compiled program that runs within R and it listens for connection requests via network sockets on a pre-defined port. This is the universal way of getting two programs communicated because the server and client programs should not be set up in same machines and compiled to run together. Although *R* does not support multi-threading internally, *Rserve* can manipulate more than one *R* processes simultaneously. The main limitation of this library is that it requires some operation system based privileges are set for network sockets.

rJava [3] is an other software library which is distributed as bundle of old *rJava* and *JRI* packages. *rJava* communicates *Java* and *R* throughout *JNI* (Java Native Interface) which is the natural way of calling compiled binary code in *Java* [10]. This library has some merit among others, in means of full interoperability between *R* and *Java* using callbacks. It is robust for large scale projects, however, it is painless at start-up process for relatively small projects. It is also been said that although *JNI* is native, it is not that fast because of type conversations between *C* and *Java* [11]. A detailed comparison of *Rserve* and *rJava* is reported in [5].

RCaller [12] is another way of calling *R* codes from within *Java*. *RCaller* converts data structures to *R* code, sends them to an externally created *R* process, returns the generated results as *XML* which is the universal way of storing data. *XML* structure is then parsed and returned values are accessed directly in *Java*. Optionally, one can create an external process for each single operation, however, this may cause a performance drawback. *RCaller* also supports sequential run of commands in a single *R* process. As it does not share the same memory area when calling external code, permits running more than one processes simultaneously and splits the running environments. RCaller has some nice features of those which its competitors already have. But the key point is simplicity. RCaller depends on a single *jar* file and no more setting up procedure is required.

3 Usage and Examples

3.1 Setup and Installation

The current version 2.2 does not require any files and ready for downloading and compiling [12]. Older versions of *RCaller* requires the R package *Runiversal* to be installed. After downloading the source tree, the *jar* file can be compiled by using *Maven* as shown below.

```
$ hg clone https://code.google.com/p/rcaller/
$ cd rcaller/RCaller
$ mvn package
```

The other option is to download pre-compiled file and it can be found at the home page of the project.

3.2 Basic Interactions

RCaller wraps complex interactions in an easy way. Since calculations are handled at *R* side, the full path to *Rscript* executable file can defined correctly using *setRscriptExecutable* method. *Java* arrays can also be passed to *R* in an easy way. Suppose that $matrix$ is a matrix with dimensions 2×2. In the example below, $matrix$ is passed to *R* and inverse of this matrix is calculated at *R* side and result is handled in *Java* again.

```
RCaller caller = new RCaller();
caller.setRscriptExecutable("path/to/Rscript.exe");

double[][] matrix = new double[][]{{6, 4}, {9, 8}};

RCode code = new RCode();

// Passing Java objects to R
code.addDoubleMatrix("x", matrix);
code.addRCode("s <- solve(x)");
caller.setRCode(code);

// Performing Calculations
caller.runAndReturnResult("s");

// Passing R object to Java
double[][] inverse = caller.getParser().getAsDoubleMatrix("s", 2, 2);
```

The interesting part of this code is the line of *caller.runAndReturnResult("s");*. The object *s* is an *R* primitive such as *vector* or *matrix* and it is returned in *XML* object. In *getAsDoubleMatrix("s", 2, 2)*, the matrix *s* is supposed to have dimensions of 2×2. In some cases, dimension vector of a returned matrix is unknown. To cope with this, *getDimension* method can be invoked as shown in the example below.

```
int[] mydim = caller.getParser().getDimensions("s");
double[][] inverse = caller.getParser().getAsDoubleMatrix("s", mydim[0], mydim[1]);
```

The content of the matrix $inverse$ is shown in Table 1.

Table 1: Content of the matrix $inverse$

	1	2
1	0.67	-0.33
2	-0.75	0.50

Objects of objects can be returned instead. Suppose that some descriptive statistics are calculated and returned to *Java*. The example below shows a calculation with multiple results.

```
code.addDoubleArray("x", new double[]{1.0, 2.2, 3.9, 4.3, 5.5});

// Returning multiple values from R to Java using lists
code.addRCode("s <- list(r1=mean(x), r2=median(x), r3=sd(x))");
caller.setRCode(code);
```

```
// Handling returned list in Java
caller.runAndReturnResult("s");

// Accessing r1, r2, r3 in Java
double mean = caller.getParser().getAsDoubleArray("r1")[0];
double median = caller.getParser().getAsDoubleArray("r2")[0];
double sd = caller.getParser().getAsDoubleArray("r3")[0];
```

In this way, results of multiple calculations can be collected in a single *list* object and external process call burden can be handled efficiently. Note that, variables $mean$,$median$ and sd hold the values 3.380000, 3.900000 and 1.779607, respectively.

3.3 Passing Plain Java Objects

Plain *Java* Objects (PJOs) can be passed to R in an efficient way. Suppose that *TestClassWithArrays* is a *Java* class which inherits an other class *TestClass*.

```
class TestClass {
  public int i = 9;
  public float f = 10.0f;
  public double d = 3.14;
  public boolean b = true;
  public String s = "test";
}

// TestClassWithArrays have variables i, f, d, b, s
// because it inherits TestClass
class TestClassWithArrays extends TestClass {
  public int[] ia = new int[]{1, 2, 3, 4, 5};
  public double[] da = new double[]{1.0, 2.0, 3.0, 4.0, 9.9, 10.1};
  public String[] sa = new String[]{"One", "Two", "Three"};
  public boolean[] ba = new boolean[]{true, true, false};
}

TestClassWithArrays tcwa = new TestClassWithArrays();

// Making a Java object 'passable' to R
JavaObject jo = new JavaObject("tcwa", tcwa);

RCaller rcaller = new RCaller();
RCode code = new RCode();

Globals.detect_current_rscript();
rcaller.setRscriptExecutable(Globals.Rscript_current);
code.clear();

// Java class is being passed to R with its members
code.addRCode(jo.produceRCode(false));

// The member 'da' of Java class 'tcwa' is accessable
```

```
// in R using tcwa$da
code.addRCode("result <- quantile(tcwa$da, 0.95)");

rcaller.setRCode(code);
rcaller.runAndReturnResult("tcwa");

double quantile = rcaller.getParser().getAsDoubleArray("result")[0];
```

In the example above, the double array *da* is a member of object *tcwa* which is accessed in *Java* using the notation *tcwa.da* is also current in the *R* side with the corresponding notation *tcwa$da*. Finally, 0.95th quantile of object *tcwa$da* is returned as 10.05.

3.4 Sequential Commands

Each time *runAndReturnResult* method invoked, a *Rscript* process is created and this operation causes a computational burden. Therefore, it is not convenient to invoke this method sequentially. Instead of *runAndReturnResult*, the method *runAndReturnResultOnline* can be used. The example below shows using a single *R* process for calculating 3 separated commands sequentially in an efficient way.

```
rcaller.setRExecutable("path/to/R.exe");

code.addDoubleArray("x", new double[]{1.0, 2.0, 3.0, 4.0, 50.0});
code.addRCode("result <- mean(x)");

// First step, getting mean of x
// Creating a single R process
rcaller.setRCode(code);
rcaller.runAndReturnResultOnline("result");
double mean = rcaller.getParser().getAsDoubleArray("result")[0];
System.out.println("mean: " + mean);

// Getting standard deviation of x through same process
code.clear();
code.addRCode("result <- sd(x)");
rcaller.runAndReturnResultOnline("result");
double sd = rcaller.getParser().getAsDoubleArray("result")[0];
System.out.println("sd: " + sd);

// Getting mad of x through same process
code.clear();
code.addRCode("result <- mad(x)");
rcaller.runAndReturnResultOnline("result");
double mad = rcaller.getParser().getAsDoubleArray("result")[0];
System.out.println("mad: " + mad);

rcaller.stopStreamConsumers()
```

In the example above, a single *R* process instead of *Rscript* process is created. It is important here to be aware of using *rcaller.setRExecutable("path/to/R.exe");* which defines the location of *R* executable. This process is first used for calculating the *mean* of x, then the result is handled in *Java*,

following this, standard deviation of x is calculated using the same process and these operations go on. Finally, after invoking the method *stopStreamConsumers*, process ends. The output is shown below.

```
mean: 12.0
sd: 21.2720473861826
mad: 1.4826
```

3.5 Generating Plots

Standard graphics routines in R such as *plot*, *lines*, *points*, etc., draws on screen by default. However, directive functions such as *png*, *bmp*, *jpeg*, *pdf*, etc., switches the device to a file with corresponding format. *RCaller* wraps these utilities in a clever way. The example given below produces a line graphics.

```
double[] numbers = new double[]{1, 4, 3, 5, 6, 10};

code.addDoubleArray("x", numbers);

// Generated plot will be handled using a File object
File file = code.startPlot();

// Generating the plot in R
code.addRCode("plot(x, pch=19)");

// Stop generating
code.endPlot();

caller.setRCode(code);

// 'file' holds the file information of generated image
// an ImageIcon can be created using 'file'
caller.runOnly();
code.showPlot(file);
```

In the example above, neither *runAndResultResult* nor *runAndReturnResultOnline* methods are used. *runOnly* method is responsible for creating an *Rscript* process and creating the graphics. *startPlot* method creates a *png* file by default and following codes create the aimed graphics. Finally, *endPlot* method closes the device. The *file* object points to created image which is accessible in *Java*. This image file can be later accessed in *ImageIcon* type as shown below

```
ImageIcon myplot = rcaller.getPlot(file);
```

and the generated plot is shown in Figure 1.

4 Performance Issues

We perform a simulation study to compare required computation times of three libraries by means of

- Establishing a connection between *Java* and *R*

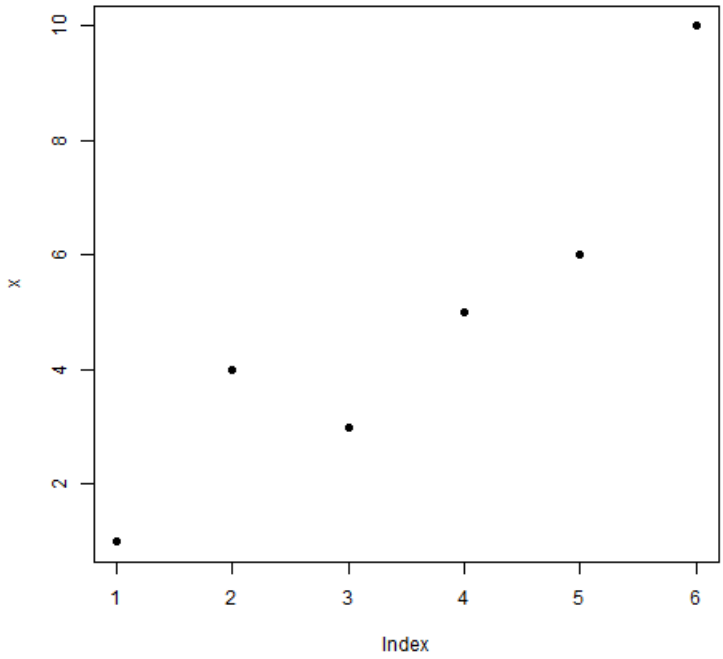

Figure 1: Generated Plot

- Passing data from *Java* to *R*
- Handling returned results from *R* to *Java*.

The test case includes *Rserve*, *rJava*, *RCaller* with the method *runAndReturnResult* and *RCaller* with the method *runAndReturnResultOnline*. Simulations are implemented and run in a PC with 4 GBs memory, a solid state disc and Linux (Ubuntu) OS installed. Setting up process is relatively easy for *RCaller* and *Rserve* as the former requires a single *jar* file added to *classpath* and the latter requires two *jar* files added and an *Rserve* process is started within R. Differently, *rJava* requires the *java.library.path* variable to be correctly defined to point out the location of *jri.dll* (libjri.so in Linux). In the test case, we create an integer array x with size of 1000. This array is then passed to *R*, x^2 values are calculated and the new array is handled by *Java*. While operations performed in *R* side do not affect the performances of libraries, no larger calculations are preferred. Operations are iterated 100 times for each library. Performances of algorithms are shown in Table 2.

In Table 2 it is shown that the *rJava* and *Rserve* outperform the RCaller by means of interaction times. Performance of *RCaller* includes creating of external *Rscript.exe* process, so the interaction time is high as it is expected. While *RCaller Online* uses single *R* process, performance increases drastically. Performances of *rJava* and *Rserve* look similar in Table 2. For these two algorithms, we test equality of location parameters of performances against to the hypothesis of in-equality of location parameters via *Mann-Whitney U* test. While the reported p-value is 0.487, we fail to reject the null hypothesis, that is, performances of *rJava* and *Rserve* are not statistically different.

Table 2: Descriptive statistics of times consumed by algorithms (in milliseconds)

	RCaller	RCaller Online	Rserve	rJava
Min	557.00	257.00	0.00	0.00
Max	643.00	296.00	14.00	9.00
Mean	569.90	266.96	1.21	1.28
Median	565.00	263.00	1.00	1.00
Std.Dev.	14.92	9.63	1.76	1.39
MAD	4.45	5.93	0.00	1.48

5 Conclusions

RCaller is a software library which is developed to simplify calling *R* from *Java*. Despite it is not the most efficient way of calling *R* codes from *Java*, it is very simple to use and its learning curve is steep. It successfully simplifies and wraps type conversations and makes variables in each languages accessible between platforms. With the calling sequential commands facility, the performance is not lost through a single external process. Although *R* is single-threaded, multiple *R* processes can be created and handled by multiple *RCaller* instances in *Java*. A *Servlet* based application can instantiate many *RCaller* objects as well as it can use a single object by using sequential command invocation ability. The former use multiple environments which do not share the same variable pool, whereas, the latter shares a mutual variable pool and clients can communicate as well. *RCaller* is written purely in *Java* and it does not depend on any external libraries, that is, it is ready to run in any machines that *Java* and *R* installed. Simulation studies show that the other libraries such as *Rserve* and *rJava* outperform the *RCaller* by means of interaction times. As a result of this, RCaller is not suitable for the projects which have many clients that request relatively single and small computations.

Competing Interests

The author declares that no competing interests exist.

References

[1] R Core Team, R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria; 2014.
Available: $http://www.R-project.org/$.

[2] Eddelbuettel D, Seamless R. C++ Integration with Rcpp. Springer, New York; 2013.

[3] Urbanek S. rJava: Low-level R to Java interface. R package version 0.9-5; 2013.
Available: $http://CRAN.R-project.org/package=rJava$.

[4] Xiao-Qin X, McClelland M, Wang Y. Pype R. A Python package for using R in Python. Journal of Statistical Software. Code Snippets. 2010;35(2): 1-8.

[5] Urbanek S. How to talk to strangers: ways to leverage connectivity between R, Java and Objective C. Computational Statistics. 2009;24(2):303-311.

[6] Gosling J, Joy B, Steele GL, Bracha G. The Java language specification. Addison-Wesley Professional; 2000.

[7] Venners B. Inside the Java virtual machine. McGraw-Hill Inc; 1996.

[8] The Apache Software Foundation. Commons-math: The apache commons mathematics library. Accessed 1 May 2014
Available: $http://commons.apache.org/math$.

[9] Urbanek S. A fast way to provide R functionality to applications. In Proceedings of DSC. 2003;2.
Available: $http://www.ci.tuwien.ac.at/Conferences/DSC-2003/Drafts/Urbanek.pdf$.

[10] Gordon R. Essential JNI: Java Native Interface. Prentice-Hall Inc; 1998.

[11] Wenner R. JNI testing - In Extreme Programming and Agile Methods-XP/Agile Universe - Springer Berlin Heidelberg. 2003;2753:96-110.

[12] R Caller Development Team. R Caller: A library for calling R from Java; 2011.
Available: $http://code.google.com/p/rcaller$.

Peer-review history:
The peer review history for this paper can be accessed here (Please copy paste the total link in your browser address bar)
www.sciencedomain.org/review-history.php?iid=550&id=6&aid=4838