Paper 4435-2020

RSUB, CLI for SAS® Server Environments

Matthew Kastin, NORC at the University of Chicago

ABSTRACT

RSUB is a command line interface, written in Java, which takes advantage of SAS Integration Technologies to fill a gap in SAS 9. NORC at the University of Chicago performed a migration from PC SAS to a clustered SAS Server Platform without SAS/Grid. This left many users accustomed to performing batch processing without their local SAS executable or alternatives provided in SAS Grid, such as bsub and sasgsub commands. Creating the RSUB utility fulfilled the needs of these users, providing them a CLI to use for batch processing and scheduling in third party enterprise schedulers.

INTRODUCTION

NORC at the University of Chicago developed the RSUB utility to meet the needs of its SAS user community. During migration from using SAS primarily on individual user workstations (e.g. PC SAS/SAS DMS) to a server environment is no small undertaking. One aspect we felt strongly about focusing on was user experience. Many SAS users at NORC had never used any of the modern GUI based interfaces, for example: SAS Enterprise Guide or SAS Studio. These users were also accustomed to performing many tasks in SAS from the command line, or similarly, through batch or PowerShell scripts or in the Windows Task Scheduler. We determined that there was no good analog for the command line processing our users would expect and endeavored to fill this gap on our own. The RSUB utility is the fruit of our labor and we are very happy to have the opportunity to share it with the SAS community at large.

This paper aims to describe the RSUB utility, its features and usage. Additionally, we will use it to explore topics common to using SAS Integration Technologies Java API for developing similar projects.

RSUB OVERVIEW

RSUB is a utility for the command line which provides an interface to the SAS system. It allows you to start a SAS session, submit a program and retrieve logging and listing outputs in return.

The RSUB utility provides interfaces for the following:

- Start a SAS session on:
  - The same host where RSUB is running
  - A remote host running a SAS Workspace Server
  - A cluster of remote hosts running SAS Workspace Servers

- A single or cluster of SAS Metadata Servers providing the connection information for a single or cluster of SAS Workspace Server(s)
- A remote host running SAS Grid Manager to connect to a SAS grid

SYSTEM REQUIREMENTS

You can connect to SAS on any platform that is supported for the specified SAS release. Older versions are likely to also work but are untested.

- SAS 9.4 or higher
- Java 8 or higher (special considerations are required for newer versions) 64-bit
- 5 jar files provided with your SAS installation
- 2 dll files provided with your SAS installation when using IWA authentication from Windows clients (local or remote)

INSTALLATION AND CONFIGURATION

We will describe installation from source only. To install from a release, simply unzip the distribution package to the desired location. Since there are dependencies that come from your SAS installation, NORC at the University of Chicago will never distribute a compiled package.

INSTALLATION

The source code for the RSUB project is available on GitHub. You are free to download, clone or fork. In order to get a copy of the source code for RSUB, you can either download it as a zip file from https://github.com/FriedEgg/rsub or you can use git to clone the repository.

```
git clone https://github.com/FriedEgg/rsub.git
```

Snippet 1. Git clone command

To work with this source code after downloading, you should use an IDE for Java, such as JetBrains IntelliJ IDEA or Microsoft Visual Studio Code.

CONFIGURATION

Now that you have the source code for the RSUB utility, you may notice that the RSUB package is actually a collection of several interconnecting modules and the CLI is just a small part of the overall capabilities we have built on top of our core module, which is merely an abstraction of the SAS Integration Technologies API for Java.

We recommend that you use Java 8, and specifically a Zulu Community JDK distribution from Azul Systems: http://azul.com. This is the same Java distribution which is included with SAS 9.4 installations.

In the rsub-iom module, there is a folder called lib (./rsub/rsub-iom/lib). This is the location where you want to put the jar files you will copy from your site's SAS 9.4 installation. These jar libraries are:

1. sas.core.jar
2. sas.rutil.jar
3. sas.security.sspi.jar
4. sas.svc.connection.jar
5. sastpj.rutil.jar

You can find these files in one (or all) of the following locations (your specific version numbers may vary):

- C:\Program Files\SASHome\SASDeploymentManager\9.4\products\deploywiz__94528__prt__xx__sp0__1\deploywiz
- C:\Program Files\SASHome\SASSecureJavaM5\9.4
- C:\Program Files\SASHome\SASVersionedJarRepository\eclipse\plugins

Next, you will want to find the lib folder inside the rsub-cli module (./rsub/rsub-cli/lib). This is the location where you want to copy the dll files from your sites SAS 9.4 installation. These dll files are:

1. sspiauth.dll
2. sspiauth_wx6.dll

These dll files are used on a Windows client when initiating either a local SAS session or a remote SAS session using Integration Windows Authentication (IWA). If you do not intend to use this feature, these files can be ignored. You can find these files in the following location:

1. C:\Program Files\SASHome\SASFoundation\9.4\core\sasext

While the RSUB utility supports a variety of connection configuration schemes, we have found that using the XML configuration is the simplest for our users. This is the method we recommend you use as well, and will be the only method we will describe in detail here. Below is an example of this file, which is also present in the source code you have cloned in the rsub-cli module root directory (./rsub/rsub-cli/serverinfo.xml). Snippet 2 shows how to setup a connection to use IWA and AES encryption of all communication between the client and server:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Redirect>
    <LogicalServer ClassIdentifier="440196d4-90f0-11d0-9f41-00a024bb830c"
                   Name="SASApp">
        <Properties>
            <Property DefaultValue="Negotiate"
                      Name="Security Package"
                      PropertyName="SecurityPackage">
            </Property>
            <Property DefaultValue="Kerberos,NTLM"
                      Name="Security Package List"
                      PropertyName="SecurityPackageList">
            </Property>
        </Properties>
        <UsingComponents>
            <ServerComponent ClassIdentifier="440196d4-90f0-11d0-9f41-00a024bb830c"
                             Name="SASApp">
                <SourceConnection>
                    <TCPIPConnection ApplicationProtocol="Bridge"
                                     CommunicationProtocol="TCP"
                                     HostName="sas01.domain.org" Name="" Port="8591">
                        <Properties>
                            <Property DefaultValue="everything"
                                      Name="Required Encryption Level"
                                      PropertyName="RequiredEncryptionLevel">
                            </Property>
                            <Property DefaultValue="AES"
                                      Name="ServerEncryptionAlgorithms"
                                      PropertyName="ServerEncryptionAlgorithms">
                            </Property>
                        </Properties>
                        <Domain>
                            <AuthenticationDomain Name="DefaultAuth"></AuthenticationDomain>
                        </Domain>
                    </TCPIPConnection>
                </SourceConnection>
            </ServerComponent>
        </UsingComponents>
    </LogicalServer>
</Redirect>
```

Snippet 2. Example serverinfo.xml file

To setup a failover cluster connection, you would simply add additional entries of the ServerComponent object for each of the additional hosts you have available.

Similar files can be generated for you by using the SAS Integration Technologies Wizard but will be for connecting to a SAS Metadata Server instead of the SAS Workspace Server. You may want to use it as a template. If you do this, it is important to reference the hosts of you SAS Workspace Servers, in place of the SAS Metadata Server hostnames, if they differ and the correct ports (8591 by default). Also important is to change the ClassIdentifier GUID to what you see in the example above, which stands for SAS Workspace Server. The wizard can also generate corresponding userinfo.xml files, which you will want to use if you need to perform Username/Password authentication instead of IWA or similar methods. Placing the userinfo.xml file in the same location as the serverinfo.xml would be the final step, in that case.

One additional configuration step we will not cover in depth, but you may find useful, is the log4j2.xml file located at (./rsub/rsub-cli/src/main/resources/log4j2.xml). This file can be manipulated to get additional feedback from the SAS API and can aide in debugging issues. The file contains the common loggers you may want additional feedback from and a console appender for them to write to. By default, these loggers will only report errors.

BUILDING

Once you have completed the configuration steps, you are ready to build the RSUB cli module. The RSUB project is designed around using the Gradle Build Tool, by Gradle Inc. This allowed us to simplify the build process and automate a number of tasks, such as the generation of batch and PowerShell scripts. To perform the build with Gradle and produce a distributable package for yourself or others in your organization, you would run the following command from the RSUB project root (./rsub):

```
gradlew :rsub-cli:distZip
```

Snippet 3. Command to run the distribution task for Gradle

This will create a distributions zip file located at (./rsub/rsub-cli/build/distributions/rsub-cli-1.1a.zip) that you can now copy anywhere, extract, and use to submit and execute SAS code and retrieve logging and listing output.

SYNTAX

Our goal was to make using RSUB from the command line as intuitive as possible by keeping what we could as similar to the batch processing our users were accustomed to. As such, we can look at the following usage text that you receive by running the rsub command with no arguments or with one of the standard help arguments (?, -h or –help). It will output the following:

```
Usage: rsub [--config-class <bridgeConfigurationClass>] [--encoding
               <encoding>] [-log destination] [-print destination] -sysin
               file-specification [-sysparm characters] [-set
               <String=String>]...
      -sysin file-specification
                              specifies a file containing a SAS program.  The
                                file-specification must be a valid Windows
                                filename.
      -log destination        specifies the destination for the SAS log.  The
                                destination must be a valid Windows filename.
      -print destination      specifies the destination for the SAS log.  The
                                destination must be a valid Windows filename.
      -sysparm characters     specifies a character string that can be passed to
                                SAS programs.
      -set <String=String>    defines an environment variable.
      --config-class <bridgeConfigurationClass>

      --encoding <encoding>
```

Snippet 4. RSUB command usage terminal output

EXAMPLES

EXAMPLE 1: SIMPLEST METHOD TO SUBMIT A PROGRAM

This example shows the simplest method to submit a program to a pre-configured installation of RSUB. The only required parameter for the rsub command is -SYSIN as shown below:

```
rsub -sysin C:\SAS\MyPrograms\test.sas
```

The above example would perform the following actions:

1. Open a new SAS session
2. Submit the code contained in the file C:\SAS\MyPrograms\test.sas to the session to run
3. Create a log file at path C:\SAS\MyPrograms\test_<datetime>.log where <datetime> would represent when the rsub command started processing your request.
4. The rsub command would wait for the completion of the code submission and then return to the command line.

EXAMPLE 2: SPECIFY LOCATIONS FOR LOGGING AND LISTING OUTPUTS

If you want to specify your own log file name and/or location, you want to use the –LOG parameter. Listing results output is not created by default. Therefore, you must also specify the –PRINT parameter with the file-specification if you want to collect them without coding the ODS LISTING statements yourself in your code. Note that in the following example the ^ character (Shift + 6) is a special character in Windows Command Prompt used to indicate line continuation. This is not a requirement and is instead done here to make the example more readable and able to be copy-pasted and still work:

```
rsub -sysin C:\SAS\MyPrograms\test.sas ^
  –log C:\SAS\MyLogs\test1.log ^
  –print C:\SAS\MyLsts\output.lst
```

The above example would perform the following actions:

1. Open a new SAS session
2. Submit the code contained in the file C:\SAS\MyPrograms\test.sas to the session to run
3. Create (and overwrite, if it exists) the log file C:\SAS\MyLogs\test1.log
4. Create (and overwrite, if it exists) the lst file C:\SAS\MyLsts\output.lst
5. The rsub command would wait for the completion of the code submission and then return to the command line.

EXAMPLE 3: A COMPLETE EXAMPLE USING ALL OF THE TYPICAL PARAMETERS

The –SYSPARM and –SET parameters are useful for passing information from the command line into your SAS programs. We recommend using -SYSPARM to pass parameters to control processing in your SAS programs and environment variables, with -SET for settings at initialization. As in the previous example, we use the ^ character in the example below to aide readability in this documents format.

```
rsub -sysin C:\SAS\MyPrograms\test.sas ^
  –log C:\SAS\MyLogs\test1.log ^
  –print C:\SAS\MyLsts\output.lst ^
  -sysparm "A 5"
  -set "ODSOUTPATH"="C:\SAS\MyLsts"
  -set "ENVFLAG"="FOO"
```

The above example would perform all of the same actions as in Example 2 plus the following actions:

1. Before the code is submitted, the environment variables and their associated values will be assigned in the connected SAS session for the multiple –SET parameters
2. Before the code is submitted and after the environment variables are assigned, the values specified in –SYSPARM will be passed to the SAS session

EXAMPLE 4: USING A ZERO CONFIGURATION LOCAL SAS SESSION

While we created RSUB with the intention to use it exclusively for connecting to remote sessions, it was useful, especially for testing, to also have the ability to run locally on demand. This is where the --CONFIG-CLASS parameter becomes useful, for example:

```
rsub -sysin C:\SAS\MyPrograms\test.sas --config-class org.norc.rsub.impl.ZeroConfiguration
```

The above is identical to the first example shared here, with the distinct difference that the first example will use the XmlConfiguration file and create a remote SAS session and this example will instead create a local SAS session.

EXAMPLE 5: USING THE RSUB-CLI JAR DIRECTLY

This example assumes far more familiarity with Java than the previous examples, but also shows some additional functionality we have not built into the rsub script files at this time because they served little utility to our users.

```
java –Djava.library.path="C:\Program Files\SASHome\SASFoundation\core\sasext" ^
  –Dlog4j.configurationFile="log4j2.xml" ^
  –Drsub.serverinfo="serverinfo.xml" ^
  –Drsub.userinfo="userinfo.xml" ^
  -classpath ".\libs" ^
  org.norc.rsub.App ^
  -sysin C:\SAS\MyPrograms\test.sas
```

The notable differences here to the previous examples is that previously, specifying the serverinfo and userinfo file locations was not possible from the command line. The serverinfo file contains the XmlConfiguration settings to connect to a remote SAS Workspace Server, and optionally, use IWA or similar authentication methods. The userinfo file allows you to specify a Username and encoded Password instead. It otherwise duplicates the functionality of the first example here.

PARAMETERS

We will first review the program parameters that are synonymous with SAS system options. The following parameters were added to rsub because they were either compulsory to the goals of rsub or appeared frequently in batch programs created by our users, and thus represented necessary features:

1. -SYSIN, expects a file-specification and denotes the SAS program file you wish to submit to the established SAS session once connected
2. -LOG, optionally expects a destination to which rsub will write the logging information generated by the SAS session. If a destination is not specified, a log is created by default which takes the file-specification from SYSIN, adds a timestamp and changes the extension to .log. The associated SAS system option –NOLOG is not supported. This was a decision we made purposefully as we always wanted a log created for all of our users and processes.
3. -PRINT, optionally expects a destination to which rsub will write the listing output generated by SAS procedures during the SAS session. If no destination is provided, then –NOPRINT is assumed and listing results will be ignored. This does not prevent a user from specifying their own listing or other ods results manually in their submitted program code.
4. -SYSPARM, optionally expects a string of characters that can be passed to SAS programs. The character string specified can be accessed in a SAS DATA step by the SYSPARM() function or anywhere in a SAS program by using the automatic macro variable referenced by &SYSPARM.
5. -SET, takes a key/value *pair variable=value* and assigns *value* to environment variable *variable* in the SAS spawned session. You specify multiple keyvalue pairs by specifying the SET parameter once per pair. The values specified can be accessed in a SAS DATA step by the SYSGET() function or anywhere in a SAS program by using the %SYSGET macro function. It is not possible to use the SET parameter to alter the assignments of items defined in server configuration files, such as the sample source library. This is intentional and meant to keep rsub mindful of multi-user environments and server administration. This parameter does function slightly differently than the SAS Foundation executable on the command line.

There are many other familiar system options that we have chosen to not implement for security purposes, or to maintain the reliability or shared systems such as: config, memsize, memlib, cpucount, work and utilloc. These are examples of options that were commonly used in our user community that we felt were not ideal to provide. Others, such as batch, noicon, nosplash and nodms are redundant given how rsub works and thus were not implemented either.

We have just reviewed the parameters of the rsub command that should be familiar to users of SAS in batch-mode and that now leaves the following parameters that are specific to RSUB:

6. --CONFIG-CLASS, is the classname from the org.norc.rsub.impl package in the rsub-iom module that you want to use to configure you SAS session connection. By default, we use org.norc.rsub.impl.XmlConfiguration, which we reviewed in the Configuration section above. Another example is the org.norc.rsub.impl.ZeroConfiguration, which is used for creating local SAS sessions and doesn't require a SAS Workspace Server to exist.
7. --ENCODING, specifies the character-encoding to use when reading the SAS code file into Java, which is then sent to SAS for processing. Typically, this should not be necessary to change. The default value is UTF-8. Changing this value does not affect the SAS session encoding.

NOTES

RETURN CODES AND COMPLETION STATUS

When running SAS in batch mode under Windows, there are six documented values for return code (or ERRORLEVEL). We condensed this simply to the three codes we cared most about discerning between:

- SUCCESS, a return code of 0 denotes that all steps in the process terminated normally.
- WARNING, a return code of 1 denotes that SAS issued warning(s)
- ERROR, a return code of 2 denotes that SAS issues error(s) or there was a User issued ABORT statement triggered (with or without RETURN/ABEND) or there was a SAS internal error.

Many of our users will further recode these exit codes and collapse WARNING as a SUCCESS. This would depend on your specific program you are running. This would not necessarily be a good practice in general.

The RSUB command assigns the return code based upon the value of SYSCC at the end of execution of the SYSIN program.

LOGGING

In the first version of RSUB all logging information was collected after the completion of the submitted code. Users complained that this prevented them from being able to observe the

job while it was in progress. This change required a lot of work to implement and still causes some bugs in especially chatty programs that produce significant amounts of log data. Occasionally, we have users see log data written out of the expected order. This is something we are working to resolve in the future.

LESSONS LEARNED

In this section we will discuss more advanced topics relating specifically to the development of RSUB and knowledge we gained that should be applicable to you when trying to create your own projects using the SAS Integration Technologies API for Java.

XMLCONFIGURATION

It is common to see examples of connecting to SAS Workspace Server from Java using the ManualConnectionFactoryConfiguration class as in the following:

```
Credential cred = new PasswordCredential("username","password","DefaultAuth");
Server server = new BridgeServer(Server.CLSID_SAS, "sas.myhost.com", 8591);
ConnectionFactoryConfiguration cxfConfig =
    new ManualConnectionFactoryConfiguration(server);
ConnectionFactoryManager manager = new ConnectionFactoryManager();
ConnectionFactoryInterface factory = cxfManager.getFactory(cxfConfig);
ConnectionInterface cx = cxf.getConnection(cred);
org.omg.CORBA.Object cxObj = cx.getObject();
IWorkspace workspace = IWorkspaceHelper.narrow(cxObj);
```

Snippet 5. ManualConnectionFactoryConfiguration usage example

The XMLConfigurationBuilder is beneficial because it removes the need to hard code the numerous server definition properties needed to connect to a SAS Server in many instances or to code it other options, such as using program arguments or system properties to avoid hard coding. You provide the XMLConfigurationBuilder instead with a serverInfoFile which can get generated using SAS Integration Technologies Configuration Wizard as we described earlier. Optionally, it can also be given a userInfoFile, which we did not describe earlier, so we will provide an example here:

```
<?xml version="1.0" encoding="UTF-8" ?>
<AuthenticationDomain Name="DefaultAuth">
  <Logins>
    <Login Name="myLogin"
           Password="{SAS002}DA9A0A5C20629B7F34D2C88A165E5530"
           UserID="username"></Login>
    </Logins>
</AuthenticationDomain>
```

Snippet 6. Example userInfo File

The passwords can be encoded using the PWENCODE Procedure. Here is a example of how we generated the password above:

```
proc pwencode in='password' method=sas002; run;
```

Snippet 7. SAS code example to encode a password

We can then take these XML files and use the XMLConfigurationBuilder, as shown in the following example, to connect to a SAS Workspace Server:

```
File serverInfo = new File("C:\path\to\serverInfo.xml");
File userInfo = new File("C:\path\to\userInfo.xml");
XMLConfigurationBuilder builder =
    new XMLConfigurationBuilder(serverInfo, userInfo);
ManualConnectionFactoryConfiguration config = builder.getConfiguration();
Credential cred = builder.getCredentialForDomain("DefaultAuth");
[…]
```

Snippet 8. XMLConfigurationBuilder Example

Past this point, there is no difference between using the XMLConfigurationBuilder and the ManualConnectionFactoryConfiguration directly.

LANGUAGESERVICE

RSUB submits sometimes large SAS programs in a single submit method call to the LanguageService. These jobs can also take extended periods of time to run. The LanguageService, by default, executes the Submit method in blocking mode, where the client holds execution until the associated program source frame has completed. We needed to execute our programs asynchronously in order to collect the log and listing output as it was generated. This means setting the Async property on the LanguageService to true. We also needed to set the attribute FlushLogPerStep to true, otherwise the log events are not triggered at a frequency high enough to provide the feedback our users expected. This attribute is not available from the base ILanguageService interface, so we needed to instantiate the ILanguageService1_1 interface. The following is a simplified excerpt of code showing what is described in this paragraph:

```
[…]
ConnectionIterface cx = factory.getConnection(credential);
IWorkspace1_1 workspace = IWorkspace_1_1Helper.narrow(cx.getObject());
ILanguageService_1_1 lang = ILanguageService_1_1Helper.narrow(workspace);
lang.Async(true);
lang.FlushLogPerStep(true);
```

Snippet 9. ILanguageService1_1 Configuration Example

The next step is to setup our event observers/listeners. There are two event interfaces we want to implement:

1. LanguageEventsListener: For the logging done in RSUB, we implemented the ILanguageLogEventOperations interface that defines two methods we care about: StepLogLines and StepListLines. These events are triggered each time the LanguageService processes a step (because we set the FlushLogPerStep attribute). A step can be thought of as a run-block of SAS code. This method gives you the ability to collect the log and/or list lines output by a program as it runs asynchronously.
2. LogEventsListener: We implemented the ILanguageEventsOperations in RSUB only to collect the return code from the SubmitComplete method. This interface, however, contains a number of interesting methods that users of SAS Enterprise Guide, for example, would be familiar with, as they watch the various steps start and complete (or error) in the status bar, for example.

These event listeners need to be registered with the language service. The following is an example of how to do that:

```
[…]
ILanguageLogEventPOATie logEventServant =
    new ILanguageLogEventPOATie(new LogEventsListener());
int logHandle = EventUtil.advise(lang, ILanguageLogEventHelper.id(), logEventServant);

ILanguageEventsPOATie langEventServant =
    new ILanguageEventsPOATie(new LanguageEventsListener());
int langHandle = EventUtil.advise(
    lang, ILanguageEventsHelper.id(), langEventServant);

[…]
Lang.Submit(program);
[…]

EventUtil.unadvise(lang, ILanguageLogEventHelper.id(), logHandle);
EventUtil.unadvise(lang, ILanguageEventsHelper.id(), langHandle);
```

Snippet 10. Example of advising ILanguageService Event Listeners

EXPERIMENTS

In the RSUB project, there are two small modules included as examples or additionally functionality we build on top of the rsub-core and rsub-iom modules (in addition to rsub-cli). These experiments have been fully implemented at NORC in some form, but are not being shared in full in the scope of this paper.

RSUB-POOL

An implementation of the Thread Pool pattern in Java to submit multiple SAS program source frames in parallel and also constrained to a number of threads with a queue. This type of processing can be beneficial if you have a need to run a long queue of steps, in parallel, without overwhelming your SAS Workspace Server(s) by opening too many sessions at once. There is potential to combine this with SAS Workspace Server Pooling to move through small repetitive tasks quickly when resetting your SAS session between tasks is not necessary.

RSUB-VERTX

Eclipse Vert.x is a tool-kit for building reactive applications on the JVM and is also a popular web framework. This example shows how to exploit the Vert.X event bus and worker verticles to perform SAS Workspace Server tasks as part of a modern web application. In this case, it is a reimagining of "The Vert.x Worker Model" by Sébastien Le Callonnec of Mastercard Developers where we have built a coffee shop. An HTTP server verticle receives the incoming orders, logs them and sends them to the event bus for processing. The SAS worker verticle then picks up the message from the event bus and processes it. In our case, this just means that SAS will sleep for a random length of time and then deliver the prepared coffee order back to the event bus and the customer.

CONCLUSION

There are many ways to reach any one solution in SAS, but it is still sometimes best to build something that suites you personally (or as an organization). Changing courses and technologies is never an easy transition and keeping this familiar is a great way to ease the tensions of a large user community. The RSUB command line application filled a gap we felt was present when moving to the SAS 9 Intelligence Platform when we were previously most comfortable with the SAS DMS User Interface and batch processing from the command line. Our aim was to create a similar experience and we generally feel like we accomplished that goal. Along the way we learned a lot about building Java applications and connecting them to SAS and we hope that in sharing this, you will too!

ACKNOWLEDGMENTS

Thanks go to the SAS user community at NORC at the University of Chicago (NSUG) for their help in developing RSUB, their feedback and patience through the planning and development. A special thanks to Joe Matise for working especially hard in this regard.

RECOMMENDED READING

- *SAS 9.4 BI API Documentation*
- *SAS® 9.4 Integration Technologies: Java Client Developer's Guide*
- *SAS® 9.4 Companion for Windows, Fifth Edition*
- https://developer.mastercard.com/blog/the-vertx-worker-model/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Matthew Kastin
NORC at the University of Chicago
friedegg@verizon.net
http://github.com/FriedEgg