Version

5.0

JEFFERSON LAB

Data Acquisition Group

Remote Execution Guide

JEFFERSON LAB DATA ACQUISITION GROUP

Remote Execution Guide

Carl Timmer
timmer@jlab.org

07-Apr-2016

12000 Jefferson Ave
Newport News, VA 23606
Phone 757.269.7100 • Fax 757.269.6248

Table of Contents

Section 1

1. Introduction

When using a collection of network-distributed computers to accomplish a single task, it can be a difficult task to initially setup the system by running the appropriate programs/software on each machine. In addition, if any processes die and need to be restarted it can be a time-consuming job to track them down and restart them. A software package called “remote execution” or “remoteExec” was developed to assist in handling this situation. This package is written in Java.

The communication backbone upon which this application is built is the cMsg publish/subscribe messaging software package. cMsg is designed to provide client programs with asynchronous publish/subscribe and synchronous peer-to-peer messaging.

The way remoteExec works is that a cMsg client program called an Executor (already written and a part of remoteExec) is initially run on each host. Although this involves some labor by necessitating someone to login to each host and running an Executor, it need only be done once. Once the necessary Executors are running, a Commander (running anywhere) can issue directives to each of the Executors which will execute the commands and return any results (see Figure 1). Any user program can be a Commander by making a few simple Java method calls with classes provided in the cMsg jar file.

Each Executor can receive messages to make it do the following:

- Execute a command, specified as a string, in a separate process and return both error and output results.
- Create a specified Java object that implements a given interface, run it as a separate thread, and return both error and output results.
- Stop specified commands.
- Stop all commands.
- Kill yourself.
- Identify yourself to those asking.

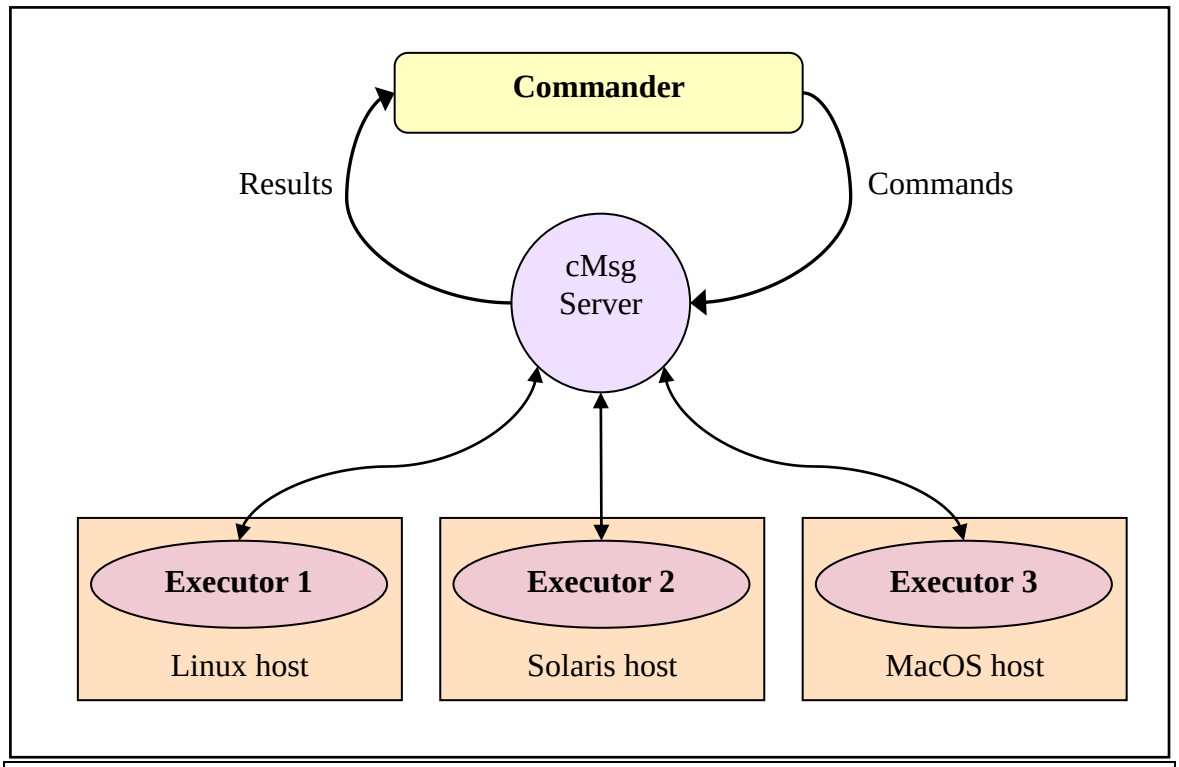

Figure 1: Basic schematic of how Commanders and Executors interact.

1.1. Purpose

The intended purpose of this software package is to be able to programmatically (inside your running program) run executable commands on other nodes and obtain the results. RemoteExec also facilitates the quick distribution of running programs. For example, the AFECS control system or a cMsg server could both be started on remote nodes by a few simple lines of code in a Commander – essentially transforming an Executor into each.

1.2. Implementation

The remoteExec package is written in the Java programming language and thus can be run wherever a Java JVM can be run. It is part of the cMsg software package with the sub-package name of **org.jlab.coda.cMsg.remoteExec**.

1.3. Prerequisites

Each running Executor must have a CLASSPATH environmental variable pointing to all necessary jar files if creating Java objects and running them as threads.

Section 2

2. Commander Basics

Any program can be a commander – that is, one that issues commands to Executors. Having just a few classes to instantiate and methods to run makes this a relatively easy chore. To start with, a Commander is created by instantiating an object of the **org.jlab.code.cMsg.remoteExec.Commander** class. There are 3 basic types of commands it can send to an Executor – those that are executable (synchronous or asynchronous) in a separate process, those that are executable as a separate thread, and those that manage the Executor itself. A full listing of available classes and their methods is given in the associated javadoc.

2.1. Running Commands as Processes

A Commander is created by instantiating an object of the **org.jlab.code.cMsg.remoteExec.Commander** class. Its constructor takes as arguments the udl enabling it to connect to a cMsg server, a unique cMsg client name, a description, and a password (if required by any Executors).

The following is a simple example with no error handling for clarity:

```
1  String cmd = "ls";
2  String udl = "cMsg://localhost/cmsg/ns";
3  String name = "me", description = "commander", password = "pswd";

4  Commander cmdr = new Commander(udl, name, description, password);
5  Collection<ExecutorInfo> execList = cmdr.getExecutors();
6  for (ExecutorInfo info : execList) {
7      CommandReturn ret = cmdr.startProcess(info, cmd, true, 12000);
8      if (ret.getOutput() != null) {
9          System.out.println("Output of " + info.getName() +":");
10         System.out.println(ret.getOutput());
11     }
12 }
```

Lines 1-3: Define some strings including the UNIX command that we will run on all Executors, ls.
Line 4: Create the Commander object. It automatically finds all Executors at the given udl.
Line 5: The list of Executors is retrieved with a single method call.
Lines 6-12: For each Executor, run the "ls" command on it, wait for it to finish, and print out the results.
Line 7: The given command is run in a separate process.

From this basic example the reader can take away a number of things. It's relatively simple to use. There is one commander object which issues commands to all Executors. Each individual command returns an object of class **org.jlab.code.cMsg.remoteExec.CommandReturn** containing, among other things, the status of that particular command and any results or errors to date. This example was of a synchronous command in which the Commander waited for each Executor output. The following is an asynchronous example of the same thing using callbacks:

```
1  String cmd = "ls";
2  String udl = "cMsg://localhost/cmsg/ns";
3  String name = "me", description = "commander", password = "pswd";

4  Class myCallback extends CommandCallback {
5      public void callback(Object userObject, CommandReturn ret) {
6          if (ret.getOutput() != null) {
7              System.out.println("Output of " + info.getName() +":");
8              System.out.println(ret.getOutput());
9          }
10     }
11 }
12 myCallback myCbk = new myCallback();

13 Commander cmdr = new Commander(udl, name, description, password);
14 Collection<ExecutorInfo> execList = cmdr.getExecutors();
15 for (ExecutorInfo info : execList) {
16     CommandReturn ret = cmdr.startProcess(info, cmd, true, myCbk, null);
17 }
```

Lines 1-3: Define some strings including the UNIX command that we will run on all Executors, ls.
Lines 4-11: Define the class containing the callback to run each time an Executor command finishes. This class must extend the CommandCallback interface which only has the one callback method in it. In this case the callback prints out the results of the executed command.
Line 12: Create an object of the myCallback class.
Line 13: Create the Commander object. It automatically finds all Executors at the given udl.
Line 14: The list of Executors is retrieved with a single method call.
Lines 15-17: For each Executor, run the "ls" command on it, and run the callback when the results are in.

Although a few lines longer than the previous example, it allows the programmer to avoid any delays due to long execution or communication times. One thing to note here is that the CommandReturn object obtained from the startProcess() method and the one passed into the callback are the very same object. The status of the command and callback can be seen by calling this object's various methods.

2.2. Running Commands in Xterms

Instead of simply running a command on an Executor, it is also possible to run this same command in an xterm which, in turn, is run by the Executor. Although this really falls under the purview of the previous section, there needs to be a few words of explanation in order to avoid problems.

Before we tackle these problems, say for example that the user wants to run a command in a terminal. The method **startXterm()** will do that and also allow use of the xterm for other purposes. Based on that method, **startWindows()** will create as many xterms as desired while arranging them neatly on the monitor. Similarly, **startCommand(s)InWindows()** will start the given command(s) in the desired number of xterms and arrange them neatly on the monitor.

Problems arise due to the nature of the xterm program and having to run it through the Java JVM. To make a long story short, any command with an argument or arguments will not be parsed correctly as the arguments will appear to be additional commands. Thus when using the methods mentioned above, only commands with no white space and therefore no arguments will be accepted. This restriction can be circumvented by placing commands into a shell script and using the script's name as the user's command.

2.3. Running Commands as Threads

In addition to running commands in a separate process, a Commander can run threads in any Java-based Executor. This may be useful if, for example, memory is limited and the user only desires one Java JVM to be running on a specific machine at any one time.

To do this, the user must provide a class, accessible to the Executor by being in its CLASSPATH, which extends the **org.jlab.code.cMsg.remoteExec.IExecutorThread** interface and its 3 methods. The first method, **startItUp(),** starts up the thread. The second method, **shutItDown()**, ends the thread. And the last method, **waitUntilDone()**, waits for the thread to finish its work which allows the Commander to do the same.

Part of the difficulty in running such a thread is creating the object of the class that implements the IExecutorThread interface to begin with. Since, in general, the constructor of such an object requires arguments, some of which are objects which themselves require constructors with args etc., it can be tricky to do this properly. In the code distribution there is a class, **org.jlab.code.cMsg.remoteExec.ExampleThread** which provides just what it says, an example of such a class.

The constructor of ExampleThread takes a java.awt.Rectangle object as the single argument which, in turn, takes a java.awt.Dimension object as an argument. To use the ExampleThread in a Commander, one would do the following:

```
15 ConstructorInfo exmplThreadCI = new ConstructorInfo();
16 ConstructorInfo rectangleCI   = new ConstructorInfo();
17 ConstructorInfo dimensionCI   = new ConstructorInfo();

18 dimensionCI.addPrimitiveArg("int", "1");
19 dimensionCI.addPrimitiveArg("int", "2");
20 rectangleCI.addReferenceArg("java.awt.Dimension", dimensionCI);
21 exmplThreadCI.addReferenceArg("java.awt.Rectangle", rectangleCI);
```

```
22 for (ExecutorInfo info : execList) {
23     CommandReturn ret = cmdr.startThread(info,
                                "org.jlab.code.cMsg.remoteExec.ExampleThread",
                                 myCbk, null, exmplThreaCI);
24 }
```

Lines 1-14: See the previous example.
Lines 15-17: A ConstructorInfo object is needed for each constructor used which has at least 1 arg.
Lines 18-19: Dimension object has a constructor which has 2 primitive integers (int) as its args. Those are specified in a distinct order with first arg being specified first, etc.
Line 20: Rectangle object has a constructor which has a Dimension object as first and only arg.
Line 21: ExampleThread object has a constructor which has a Rectangle object as first and only arg.
Lines 22-24: For each Executor, start a thread using the given class and constructor info. Run the given callback when the results are in.

2.4. Managing the Executor

2.4.1. Identifying itself

An Executor does nobody any good if Commanders don't know about it. When creating a Commander, its constructor will issue a command to all Executors using the same cMsg udl to identify themselves. The user can also tell Executors to identify themselves explicitly by calling Commander.findExecutors().

2.4.2. Stopping a Specific Process or Thread

A Commander can stop a process or thread in the middle of its execution. In general, stopping a **process** can be done immediately, but how a **thread** is stopped will actually depend on how its shutItDown() method was programmed.

Either can be stopped by calling the CommandReturn.stop() method of the CommandReturn object obtained by the initial call to Commander.startProcess() or startThread(). Another way to do the same thing is by calling the Commander.stop() method which requires arguments specifying the Executor and the id of the process or thread it is trying to stop.

```
1  CommandReturn ret = cmdr.startProcess(info, cmd, true, myCbk, null);
2  ret.stop();
3  cmdr.stop(info, ret.getId());
```

Line 1: Commander sends a command to an Executor which returns a CommandReturn object.
Line 2: Stop the given command using the CommandReturn object.
Line 3: Stop the given command using the Commander object.

2.4.3. Stopping all Processes and Threads

A Commander can stop all processes and threads on a specific Executor at once by calling Commander.stopAll() and giving the Executor as an argument. Or the Commander can stop all processes and threads on all Executors by calling the same (overloaded) method with no arguments.

2.4.4. Killing Executors

A Commander can kill a specific Executor process by calling Commander.kill() and giving the Executor as an argument. Or the Commander can kill all Executors by calling Commander.killAll(). Both methods have the option of whether to kill the running processes as well.

2.5. Commander Code Examples

Examples of Commander code doing various tasks are located that the end of the Commander.java file. There are examples of:

- giving a single command to all Executors
- starting up 20 xterms from one Executor and shutting them down one-by-one
- running an ExampleThread object as a thread in 1 Executor, waiting, shutting it down
- running a cMsg server as a thread in 1 Executor, waiting, and then shutting it down
- killing all Executors and their spawned processes either one-by-one or all at once
- taking keyboard input, passing it as commands to 1 Executor, and printing results - like a simple, remote terminal.

These examples are a good place for the user to start and can be easily expanded upon.

Section 3

3. Executor Basics

3.1. Executor

The Executor is written in the Java programming language and can be run wherever a Java virtual machine (JVM) can be run The Executor class is executable with the following command line options:

Command line option	Description
-n <name>	Name used to connect to the cMsg server specified by udl
-u <udl>	Udl used to connect to the cMsg server
-p <password>	Password a Commander must provide to access Executor

Thus to run an Executor a user would login in to a machine and type something like:

```
java Executor –n myName –u cMsg://cmsgHost/cMsg/myNamespace –p myPassword
```

Note that specifying the name is optional and will default to the host name if none is given. The password is also optional but strongly recommended to be provided. The udl is necessary, however, for the Executor to function.

3.2. Security

The astute reader (I hope you are one of them) will by now have realized that this software package, while powerful, can be used to cause harm by a malicious programmer. If as a Commander I can run any code on an Executor, I could do some real damage. That is why the use of passwords is so important. Running each Executor with a secure password is extremely critical. This requires all Commanders to provide the same password to the Executor before any command will be carried out.

Let the reader rest assured that passwords are encrypted by a 128-bit AES algorithm and then converted to text using the base64 standard before being transmitted between Commander and Executor.