Rappture Integration with Submit

Overview

It is possible to use the submit command to execute simulation jobs generated by Rappture interfaces remotely. A common approach is to create a shell script which can exec'd or forked from an application wrapper script. This approach has been applied to TCL, Python, Perl wrapper scripts. To avoid consumption of large quantities of remote resources it is imperative that the submit command be terminated when directed to do so by the application user (Abort button).

TCL Wrapper Script

submit can be called from a TCL Rappture wrapper script for remote batch job submission. An example of what code to insert in the wrapper script is detailed here.

An initial code segment is required to catch the Abort button interrupt. Setting execctl to 1 will terminate the process and any child processes.

```
package require Rappture
Rappture::signal SIGHUP sHUP {
   puts "Caught SIGHUP"
   set execctl 1
}
Rappture::signal SIGTERM sTERM {
   puts "Caught SIGTERM"
   set execctl 1
}
```

A second code segment is used to build an executable script that can executed using Rappture::exec. The trap statement will catch the interrupt thrown when the wrapper script execution is Aborted. Putting the submit command in the background allows for the possibility of issuing multiple submit commands from the script. The wait statement forces the shell script to wait for all submit commands to terminate before exiting.

```
   set    submitScript "#!/bin/sh\\n\\n"
   append submitScript "trap cleanup HUP INT QUIT ABRT TERM\\n\\n"
   append submitScript "cleanup()\\n"
   append submitScript "{\\n"
   append submitScript "   kill -TERM `jobs -p`\\n"
   append submitScript "   exit 1\\n"
```

```
   append submitScript "}\\n\\n"

   append submitScript "cd [pwd]\\n"
   append submitScript "submit -v cluster -n $nodes -w $walltime\\\\\\
n"
   append submitScript "       COMMAND ARGUMENTS &\\n"
   append submitScript "sleep 5\\n"
   append submitScript "wait\\n"

   set submitScriptPath [file join [pwd] submit_script.sh]
   set fid [open $submitScriptPath w]
   puts $fid $submitScript
   close $fid
   file attributes $submitScriptPath -permissions 00755
```

The standard method for wrapper script execution of commands can now be used. This will stream the output from all submit commands contained in submit_script.sh to the GUI display. The same output will be retained in the variable out.

```
set status [catch {Rappture::exec $submitScriptPath} out]
```

Each submit command creates files to hold COMMAND standard output and standard error. The file names are of the form JOBID.stdout and JOBID.stderr, where JOBID is an 8 digit number. These results can be gathered as follows.

```
   set out2 ""
   foreach errfile [glob -nocomplain *.stderr] {
      if [file size $errfile] {
         if {[catch {open $errfile r} fid] == 0} {
            set info [read $fid]
            close $fid
            append out2 $info
         }
      }
      file delete -force $errfile
   }
   foreach outfile [glob -nocomplain *.stdout] {
      if [file size $outfile] {
         if {[catch {open $outfile r} fid] == 0} {
            set info [read $fid]
            close $fid
            append out2 $info
         }
      }
```

```
      file delete -force $outfile
   }
```

The script file should be removed.

```
file delete -force $submitScriptPath
```

The output is presented as the job output log.

```
$driver put output.log $out2
```

All other result processing can proceed as normal.

Python Wrapper Script

submit can be called from a python Rappture wrapper script for remote batch job submission. An example of what code to insert in the wrapper script is detailed here.

An initial code segment is required to import some predefined functions that manage typical aspects of remote submission. An important aspect is the handling of user interruption via the Abort button.

```
from Rappture.tools import getCommandOutput as RapptureExec
```

A second code segment is used to build an array containing arguments to a submit command. The arguments correspond directly to what would entered at command line.

```
   submitCommand = ["submit"]
   submitCommand.append["-v"]
   submitCOmmand.append[venue]
   submitCommand.append["-n"]
   submitCommand.append["%s" % (str(nodes))]
   submitCommand.append["-w"]
   submitCommand.append["%s" % (str(walltime))]
   submitCommand.append[COMMAND]
   for commandArgument in ARGUMENTS:
      submitCommand.append[commandArgument]
```

The standard method for wrapper script execution of commands can now be used. This will stream the output from all submit commands contained in submit_script.sh to the GUI display. The same output will be retained in the variable stdOutput.

```
exitStatus,stdOutput,stdError = RapptureExec(submitCommand)
```

Each submit command creates files to hold COMMAND standard output and standard error. The file names are of the form JOBID.stdout and JOBID.stderr, where JOBID is an 8 digit number. These results can be gathered as follows.

```
reStdout = re.compile(".*.stdout$")
reStderr = re.compile(".*.stderr$")

out2 = ""
errFiles = filter(reStderr.search,os.listdir(os.getpwd()))
if errFiles != []:
   for errFile in errFiles:
      errFilePath = os.path.join(os.getpwd(),errFile)
      if os.path.getsize(errFilePath) > 0:
         f = open(errFilePath,'r')
         outFileLines = f.readlines()
         f.close()
         stderror = ''.join(outFileLines)
         out2 += 'n' + stderror
      os.remove(errFilePath)

outFiles = filter(reStdout.search,os.listdir(os.getpwd()))
if outFiles != []:
   for outFile in outFiles:
      outFilePath = os.path.join(os.getpwd(),outFile)
      if os.path.getsize(outFilePath) > 0:
         f = open(outFilePath,'r')
         outFileLines = f.readlines()
         f.close()
         stdoutput = ''.join(outFileLines)
         out2 += 'n' + stdoutput
      os.remove(outFilePath)
```

The output is presented as the job output log.

```
   lib.put("output.log", out2, append=1)
```

All other result processing can proceed as normal.

Perl Wrapper

submit can be called from a perl Rappture wrapper script for remote batch job submission. An example of what code to insert in the wrapper script is detailed here.

An initial code segment is required to catch the Abort button interrupt.

```
use Rappture

my $ChildPID = 0;

sub trapSig {
    print "Signal @_ trapped\\n";
    if($ChildPID != 0) {
        kill 'TERM', $ChildPID;
        exit 1;
    }
}
$SIG{TERM} = \&trapSig;
$SIG{HUP}  = \&trapSig;
$SIG{INT}  = \&trapSig;
```

A second code segment is used to build an executable script that can executed using Rappture.tools.getCommandOutput. The trap statement will catch the interrupt thrown when the wrapper script execution is Aborted. The wait statement forces the shell script to wait for the submit command to terminate before exiting.

```
$SCRPT = "submit_app.sh";
open(FID,">$SCRPT");
print FID "#!/bin/sh\\n";
print FID "\\n";
```

```
print FID "trap cleanup HUP INT QUIT ABRT TERM\\n\\n";
print FID "cleanup()\\n";
print FID "{\\n";
print FID "   kill -s TERM `jobs -p`\\n";
print FID "   exit 1\\n";
print FID "}\\n\\n";

print FID "submit -v cluster -n $nPROCS -w $wallTime COMMAND ARGUMENTS
 &\\n";
print FID "wait %1\\n";
print FID "exitStatus=\\$?\\n";
print FID "exit \\$exitStatus\\n";
close(FID);
chmod 0775, $SCRPT;
```

The standard fork and exec method for wrapper script execution of commands can now be used. Using this approach does not allow streaming of the command outputs.

```
if      (!defined($ChildPID = fork())) {
    die "cannot fork: $!";
} elsif ($ChildPID == 0) {
    exec("./$SCRPT") or die "cannot exec $SCRPT: $!";
    exit(0);
} else {
    waitpid($ChildPID,0);
}
```

Each submit command creates files to hold COMMAND standard output and standard error. The file names are of the form JOBID.stdout and JOBID.stderr, where JOBID is an 8 digit number. These results can be gathered with standard perl commands for file matching, reading, etc. All other result processing can proceed as normal.