Tool Developers

Learn how to create new simulation and modeling tools and publish them on a HUB. Sections include:

- Overview of Tool Development Process
- Using Subversion Source Code Control
- Rappture Toolkit for Creating Graphical User Interfaces
- Rappture web site

Overview

Tool Development Process

Each hub relies on its user community to upload tools and other resources. Hubs are normally configured to allow any user to upload a tool. The process starts with a particular user filling out a web form to register his intent to submit a tool. This tells the hub manager to create a new project area for the tool. The user then uploads code into a Subversion source code repository, and develops the code within a workspace. The user can work alone or with a team of other users. When the tool is ready for testing, the hub manager installs the tool and asks the development team to approve it. Then, the hub manager takes one last look at the tool, and if everything looks good, moves the tool to the "published" state. Of course, a tool can be improved even after it is published, and re-installed, approved, and published over and over again.

The complete process is explained in the tool maintenance documentation for hub managers. Additional details about this process can be found in the following seminars:

- Overview of Tool Development Process
- Using Workspaces
- Using Subversion for Source Code Control

Creating Graphical User Interfaces

If a tool already has a graphical user interface that runs under Linux/X11, then it can be published as-is, usually in a matter of hours. There are two caveats:

- **If the tool relies heavily on graphics, it may not perform very well within HUBzero execution containers.** Our containers run in cluster nodes without graphics cards, and are therefore configured with MESA for software emulation of OpenGL. This has much poorer performance than ordinary desktop computers with a decent graphics card, so frame rates are much lower. Also, all graphics are transmitted to the user's web browser after rendering, again lowering the frame rate. You can expect to achieve a few frames per second in the hub environment--good enough to view and interact with the data, but far below 100 frames/sec that you would normally see on a desktop computer.
- **Tools running within the hub have access to the hub's local file system--not the user's desktop.** Many tools have a *File* menu with an *Open* option. When a user invokes this option within the hub environment, it will bring up a file dialog showing the hub file system. The user won't see his own local files there unless he uploads them first via sftp, webdav, or the hub's importfile command.

The graphical user interface for any tool published in the hub environment can be created using standard toolkits for desktop applications--including Java, Matlab, Python/QT, etc.

If you're looking for an easy way to create a graphical interface for a legacy tool or simple modeling code, check out the Rappture Toolkit that is included as part of HUBzero. Rappture reads a simple XML-based description of a tool and generates a graphical user interface automatically. It interfaces naturally with many programming languages, including C/C++, Fortran, Matlab, Python, Perl, Tcl/Tk, and Ruby. It creates tools that look something like the following:

Rappture was designed for the hub environment and therefore addresses the caveats listed above. All Rappture-based tools have integrated visualization capabilities that take advantage of hardware-accelerated rendering available on the HUBzero rendering farm. Rappture-based tools also include options to upload/download data from the end user's desktop via the importfile/exportfile commands available within HUBzero.

For more details about Rappture, see the following links:

- Rappture Quick Overview
- Developing Scientific Tools for the HUBzero Platform (introductory course with 7 lectures)
- Rappture Reference Manual

Combining Tools

Overview

Some of the tools on any hub are really a collection of 3-5 programs acting like a "workbench" for a particular application. Berkeley Computational Nanoscience Class Tools is one such example. It is really a collection of several separate Rappture-based applications, all running on the same desktop, in the same tool session.

We've created a simple window manager called **nanoWhim** that makes it easy to switch back and forth between several applications on a desktop--without all of the fuss and bother associated with a typical window manager. A tool using nanoWhim looks like this:

The combobox at the top lets users switch between applications. Each window that pops up within an application is managed by a set of tabs.

nanoWhim is based on the Whim window manager written in Tcl/Tk. We needed something like this for nanoHUB to create a very simple tabbed interface, so users could easily switch between a couple of tools within the same tool session. A more comprehensive workflow interface is under development, but this simple solution is sometimes useful.

Flipping between tools

The following example shows a Rappture-based application that popped up a separate Jmol application for molecular visualization. Jmol pops up in its own tab, and you can easily switch back and forth between the original application and the Jmol popup by clicking on the tabs, as shown below:

You can click on the **x** on the Jmol tab to close that application.

You can select another application by using the combobox at the very top of the window. That brings up another Rappture-based application, with a different set of inputs and outputs.

You can run each program independently, and the outputs stay separate. If you flip back to the previous application, it will be sitting just the way you left it.

Configuring nanoWhim

To use nanoWhim, you'll need to create two files in the "middleware" directory for your tool: **nanowhimrc** and **invoke**.

The nanowhimrc File

This file configures the various applications that pop up within the tool session. Here's a very simple example:

```
# set an icon
set.config controls_icon header.gif

# first app is an xterm
start.app "Terminal Window" xterm

# second app is a web browser
start.app "Web Browser" firefox
```

Any line that starts with a pound sign (#) is treated as a comment.

The set.config command configures various aspects of the window manager. Right now, the only useful option is controls_icon, which sets the icon shown in the top-left corner of the window. Note that a relative file name is interpreted with respect to the location of the nanowhimrc file itself. In this case, we've assumed that the image header.gif is sitting in the same directory as nanowhimrc.

The rest of the file contains a series of start.app commands for each application that you want to offer. In this case, the first application is called "Terminal Window" and is just an xterm application. The second application is the Firefox web browser, which we label "Web Browser".

Here's a more realistic example:

```
#
# Customize the nanoWhim window manager
#
set.config controls_icon header.gif
```

```
start.app "Average"
  /apps/rappture/invoke_app -t ucb_compnano -T $dir/../rappture/avg -p
 /apps/java/bin

start.app "Molecular Dynamics (Lennard-Jones)"
  /apps/rappture/invoke_app -t ucb_compnano -T $dir/../rappture/ljmd -
p /apps/java/bin

start.app "Molecular Dynamics (LAMMPS)"
  /apps/rappture/invoke_app -t ucb_compnano -T $dir/../rappture/lammps
 -p /apps/java/bin -p /apps/lammps/lammps-12Feb07/bin

start.app "Monte Carlo (Hard Sphere)"
  /apps/rappture/invoke_app -t ucb_compnano -T $dir/../rappture/hsmc -
p /apps/java/bin

start.app "Ising Simulations"
  /apps/java/bin/java -classpath $dir/../bin MonteCarlo
```

Each start.app command starts a different Rappture-based application. The first argument in quotes is the title of the application, which is displayed in the combobox at the top of the window. The remaining arguments are treated as the Unix command that is invoked to start the application.

The commands shown here all use the /apps/rappture/invoke_app script to invoke a Rappture-based application. The -t argument for that script indicates the project (tool) name. The -T argument indicates which directory contains the Rappture tool.xml file. You can use $dir here to locate the directory relative to the nanowhimrc file. Each -p argument adds a directory onto the execution path (environment variable $PATH), which may be needed for simulators and other tools invoked by the Rappture program.

The invoke File

The nanowhimrc file configures the window manager, but the middleware/invoke script actually invokes it. Every tool on nanoHUB has its own invoke script sitting in the middleware directory. Your invoke script should look like this if you want to use nanoWhim:

```
#!/bin/sh
/apps/nanowhim/invoke_app -t ucb_compnano
```

This script invokes the nanoWhim window manager for the project specified by the -t argument.

This is the short name that you gave when you registered your tool with nanoHUB. This script looks for the middleware/nanowhimrc file within your source code, and launches nanoWhim with that configuration.

Testing Your Tool

Normally, you develop and test tools within a workspace in your hub. If you're using nanoWhim, that's still true for the individual applications. In other words, you can test each application individually within a workspace. But to get the full effect of the nanoWhim manager running all applications at once, you'll have to get your tool to "installed" status, and then launch the application in test mode. For details about doing this, see the tool maintenance documentation for hub managers or the lecture on Uploading and Publishing New Tools. Look at the tool status page for your own tool project and find the *Launch Tool* button. This is what you would normally do to test any tool before approving it. Once you're in the "installed" stage and you're able to click *Launch Tool*, the nanoWhim configuration should take effect and you'll be able to test the overall combined tool.

Accessing the Grid

Overview

Tools are hosted within a "tool session" running within the hub environment. The tool session supports the graphical interface, which helps the user set up the problem and visualize results. If the underlying calculation is fairly light weight (e.g., runs in a few minutes or less), then it can run right within the same tool session. But if the job is more demanding, it can be shipped off to another machine via the "submit" command, leaving the tool session host less taxed and more responsive.

This chapter describes the "submit" command, showing how it can be used at the command line within a workspace and also within Rappture-based tools.

Submit Command

Overview

submit takes a user command and executes it remotely. The objective is to allow the user to issue a command in the same manner as a locally executed command. Multiple submission mechanisms are available for job dissemination. A set of steps are executed for each job submission:

- Destination site is selected
- A wrapper script is generated for remote execution
- If needed a batch system description file is generated.
- Input files for a job are gathered and transferred to the remote site. Transferred files include the wrapper script and batch description scripts.
- The wrapper script is executed remotely.
- Progress of the remote job is monitored until completion.
- Output files from the job are returned to the dissemination point.

Command Syntax

submit command options can be determined by using the help parameter of the submit command.

```
$ submit --help
usage: submit [options]

options:
  -v, --venue           Remote job destination
  -i, --inputfile       Input file
  -n NCPUS, --nCpus=NCPUS
                        Number of processors for MPI execution
  -N PPN, --ppn=PPN     Number of processors/node for MPI execution
  -w WALLTIME, --wallTime=WALLTIME
                        Estimated walltime hh:mm:ss or minutes
  -e, --env             Variable=value
  -m, --manager         Multiprocessor job manager
  -h, --help            Report command usage

Currently available DESTINATIONs are:
   clusterA
   clusterB

Currently available MANAGERs are:
   mpich-intel32
```

By specifying a suitable set of command line parameters it is possible to execute commands on configured remote systems. The simple premise is that a typical command line can be prefaced by submit and its arguments to execute the command remotely.

```
$ submit -v clusterA echo Hello world!
Hello world!
```

In this example the echo command is executed on the venue named clusterA where jobs are executed directly on the host. Execution of the same command on a cluster using PBS would be done in a similar fashion

```
$ submit -v clusterB echo Hello world!
(2586337) Simulation Queued Wed Oct  7 14:45:21 2009
(2586337) Simulation Done Wed Oct  7 14:54:36 2009
$ cat 00577296.stdout
Hello world!
```

submit supports an extensible variety of submission mechanisms. HUBzero supported submission mechanisms are

- local - use job submission mechanisms available directly on the submit host. These include PBS and condor job submission.
- ssh - direct use of ssh with pre-generated key.
- ssh + remote batch job submission - use ssh to do batch job submission remotely, again with pre-generated key.

A site for remote submission occurs is selected in one of the following ways, listed in order of precedence:

- User specified on the command line with -v/--venue option.
- Randomly selected from remote sites associated pre-staged application.

Any files specified by the user plus internally generated scripts are packed into a tarball for delivery to the remote site. Individual files or entire directory trees may be listed as command inputs using the -i/--inputfile option. Additionally command arguments that

exist as files or directories will be packed into the tarball. If using ssh based submission mechanisms the tarball is transferred using scp.

The job wrapper script is executed remotely either directly or as a batch job. The job is subject to all remote queuing restrictions and idiosyncrasies.

Remote batch jobs are monitored for progress. Changes in job status are reported at least every minute. Job status is reported at least every four minutes. The job status is used to detect job completion.

The same methods used to transfer input files are applied in reverse to retrieve output files. Any files and directories created or modified by the application are be retrieved. A tarball is retrieved and expanded to the home base directory. It is up to the user to avoid the overwriting of files.

In addition to the application generated output files additional files are generated in the course of remote job execution. Some of these files are for internal bookkeeping and are consumed by submit, a few files however remain in the home base directory. The remaining files include JOBID.stdout and JOBID.stderr, it is also possible that a second set of standard output/error files will exist containing the output from the batch job submission script. JOBID represents unique job identifier assigned by submit.

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 RapptureGUI
Rappture::signal SIGHUP sHUP {
   puts "Caught SIGHUP"
   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 catch the Abort button interrupt.

```
import os
import stat
import Rappture
import signal

def sig_handler(signalNumber, frame):
   if Rappture.tools.commandPid > 0:
      os.kill(Rappture.tools.commandPid,signal.SIGTERM)

signal.signal(signal.SIGINT, sig_handler)
signal.signal(signal.SIGHUP, sig_handler)
signal.signal(signal.SIGQUIT, sig_handler)
signal.signal(signal.SIGABRT, sig_handler)
signal.signal(signal.SIGTERM, sig_handler)
```

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. 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.

```
   submitScriptName = 'submit_app.sh'
   submitScript     = """#!/bin/sh

trap cleanup HUP INT QUIT ABRT TERM
```

```
cleanup()
{
   echo "Abnormal termination by signal"
   kill -s TERM `jobs -p`
   exit 1
}

"""
   submitScript += "cd %s\\\n" % (os.getcwd())
   submitScript += "submit -v cluster -n %s -w %s \\\n" % (nodes,wallt
ime)
   submitScript += "       %s %s &\\\n" % (COMMAND,ARGUMENTS)
   submitScript += "wait\\\n"

   submitScriptPath = os.path.join(os.getcwd(),submitScriptName)
   fp = open(submitScriptPath,'w')
   if fp:
      fp.write(submitScript)
      fp.close()

   os.chmod(submitScriptPath,
            stat.S_IRWXU|stat.S_IRGRP|stat.S_IXGRP|stat.S_IROTH|stat.S
_IXOTH)
```

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 =
               Rappture.tools.getCommandOutput(submitScriptPath)
```

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.

```
   re_stdout = re.compile(".*\.stdout$")
   re_stderr = re.compile(".*\.stderr$")

   out2 = ""
   errFiles = filter(re_stderr.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(re_stdout.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 script file should be removed.

```
   os.remove(submitScriptPath)
```

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.