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:

- Bootcamp Course for New Developers
- 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.

TOOL DEVELOPERS

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

Invoking tools with invoke scripts

Overview

Invoke scripts are small programs, usually written in sh or bash, used to setup the application container environment so the tool can run properly. More specifically, invoke scripts are responsible for:

- Locating tool.xml for Rappture applications
- Setting up the PATH and other optional environment variables
- Starting the window manager
- Starting optional subprograms, like filexfer
- Starting the application

For most applications, the invoke script is a single command that calls the default HUBzero invoke script, named invoke_app, with a few options set. In some rare situations, the tool needs the application container setup in a manner that invoke_app cannot handle. In these cases, the tool developer can modify the tool's invoke script to appropriately setup the application container.

The sections below list out details regarding the options of invoke_app, how to launch Rappture tools using an invoke script that calls invoke_app, and how to launch non-Rappture tools using an invoke script that calls invoke_app.

invoke_app and its options

HUBZero's default tool invocation script is called invoke_app. It is a bash script, usually located in /apps/invoke/current. When called with no options, the script tries to automatically find the needed information to start the applications. There are a number of options that can be provided to alter the script's behavior.

invoke_app accepts the following options:

```
 -A tool arguments
 -c execute command in background
 -C command to execute for starting the tool
 -e environment variable (${VERSION} substituted with $TOOL_VERSION)
 -f No FULLSCREEN
 -p add to path          (${VERSION} substituted with $TOOL_VERSION)
 -r rappture version
 -t tool name
 -T tool root directory
 -u use envionment packages
 -v visualization server version
 -w specify alternate window manager
```

Here is a detailed description of the options:

-A	pass the provided enquoted arguments onto the tool. Example usage: `-A "-q blah1 -w blah2"` The options -q and -w are not parsed by invoke, but are passed on to the tool
-c	Commands to run in the background before the tool launches. Exmple usage: `-c "echo hi" -c "filexfer"` This prints "hi" to stdout and starts filexfer

-C	Command to execute for starting tool. Tool's command line arguments can be included in this option, or can be placed in the -A option.
	
	Example usage:
	
	Call a program, named myprog, located in the tool's bin directory:
	
	`-C @tool/bin/myprog`
	
	Call a program, named myprog, located in the tool's bin directory, with program arguments "-e val1" and "-b val2":
	
	`-C "@tool/bin/myprog -e val1 -b val2"`

	Call a program, named myprog, located in the tool's bin directory with arguments -e val1 and -b val2, used in conjunction with invoke_app's -A option:
	
	`-C @tool/bin/myprog -A "-e val1 -b val2"`
	
	Call a program, named myprog, located in the tool's bin directory. We can omit the path of the program if it is an executable and located in the tool's bin directory because the tool's bin directory is added to the PATH environment variable. This would not work for calling a Perl script in a fashion similar to **perl myscript.pl** because in this case, **perl** is executable and **myscript.pl** is the argument.:
	
	`-C myprog`
	
	Call simsim with no arguments:
	
	`-C /apps/rappture/bin/simsim`
	
	Call simsim with the options -tool and -values, to be parsed by simsim:
	
	`-C "/apps/rappture/bin/simsim -tool driver.xml -values random"`
	
	Call simsim with the options -tool and -values, to be parsed by simsim:
	
	`-C /apps/rappture/bin/simsim -A "-tool driver.xml -values random"`

-e	Set an environment variable. Example usage:

	`-e LD_LIBRARY_PATH=@tool/../${VERSION}/lib:${LD_LIBRARY_PATH}` Within the value part of this option's argument, the text ${VERSION} is automatically substituted with the value of the variable ${TOOL_VERSION}. Similarly, the text @tool is substituted with the value of ${TOOLDIR}. By setting the environment variable, you are overwritting its previous value.
-f	no full screen - disable FULLSCREEN environment variable, used by Rappture, to expand the window to the full available size of the screen.
-p	Prepend to the PATH environment variable. Example usage: `-p @tool/../${VERSION}/bin` Within the value part of this option's argument, the text ${VERSION} is automatically substituted with the value of the variable ${TOOL_VERSION}. Similarly, the text @tool is substituted with the value of ${TOOLDIR}. By setting this option the PATH environment variable is adjusted, but not overwritten.
-r	sets RAPPTURE_VERSION which dictates which version of rappture is used and may manipulate the version of the tool that is run. If left blank, the version will be determined by looking at $SESSIONDIR/resources file. Accpetable values include "test", "current", "dev". When RAPPTURE_VERSION is "test", RAPPTURE_VERSION is reset to current and TOOL_VERSION is set to dev. The current version of rappture is used and the dev version of the tool is used when launching the program. When RAPPTURE_VERSION is "current",

	TOOL_VERSION is set to "current". The current version of rappture is used and the current version of the tool is used when launching the program. When RAPPTURE_VERSION is "dev", TOOL_VERSION is set to "dev". The dev version of rappture is used and the dev version of the tool is used when launching the tool.
-t	sets ${toolname} which is used while setting up tool paths for TOOLDIR and TOOLXML. ${toolname} is the short name (or project name) of the tool. It is the same as the name used in the source code repository. With respect to the tool contribution process, it is the "toolname" in the path /apps/toolname/version/rappture/tool.xml. Setting this option will change the paths searched while trying to locate tool.xml and the bin directory.
-T	Tool root directory. This is the directory holding a checked out version of the code from the source code repository. It typically has the src, bin, middleware, rappture, docs, data, and examples directories underneath it. With respect to the tool contribution process, it is the "/apps/toolname/version" in the path /apps/toolname/version/rappture/tool.xml. Setting this option will change the paths searched while trying to locate tool.xml and the bin directory. Typically when testing this option is used to specify where the tool directory is. In this case, its the present working directory: `-T $PWD`
-u	Set use scripts to invoke before running the tool. Example usage: `-u octave-3.2.4 -u petsc-3.1-real-gnu` These would setup octave-3.2.4 and petsc-3.1 in the environment that your tool would launch in.

-v	Visualization server version. This option changes which visualization servers are setup in the file $SESSIONDIR/resouces. Currently, the only recognized option is dev. If left blank this option defaults to the "current" visualization servers. This option essentially decides whether to run the script update_vis or update_viz_dev. Example: `-v dev` This option irrelevant if no visualization server is available.

-w	set the window manager. The default value is to use the ratpoison window manager if it exists. If ratpoison is not installed on the system, look for the icewm captive window manager setup. Use this flag to choose an alternative window manager. Valid values for this option include: "ratpoison" and "captive"
	
	Examples:
	
	Use the icewm captive window manager.
	
	`-w captive`
	
	Use the ratpoison window manager.
	
	`-w ratpoison`

invoke_app is called from within a tool's invoke script. The invoke script is stored in the middleware directory of the tool's source code repository.

Using invoke_app with Rappture tools

Invoke scripts should be placed in the middleware directory of the tool's source code repository. A typical invoke script for a Rappture application looks similar to this:

```
#!/bin/sh

/apps/invoke/current/invoke_app "$@" \\
    -t calc
```

In the invoke script above, invoke_app, located in the directory /apps/invoke/current, is called with "$@" and "-t calc". "$@" represents all options that the invoke script itself received. "-t calc" tells invoke_app that the toolname is "calc". This information is used by invoke_app to figure out which tool it is supposed to be launching and where that tool is installed.

For most Rappture applications, the invoke script is very simple. The above is enough for invoke_app to start looking for a tool.xml file. invoke_app looks for the file named tool.xml. It uses the TOOLDIR variable to help decide where to look. If the tool.xml file is not found in the ${TOOLDIR}/rappture directory, invoke_app will exit explaining that it could not find the tool.xml file. The TOOLDIR variable can be set from the command line using the -T flag:

```
/apps/invoke/current/invoke_app "$@" -t calc -T ${PWD}
```

Actually, it is more common to see the -T flag provided to a tool's invoke script, and the option is forwarded to invoke_app by "$@":

```
./middleware/invoke -T ${PWD}
```

In the above example, the TOOLDIR variable is set to the present working directory, which is stored in the variable PWD. Specifying the -T option is usually not needed, but can help when invoke_app is confused on what it is supposed to be launching.

Using invoke_app with non-Rappture tools

Invoke scripts should be placed in the middleware directory of the tool's source code repository. A typical invoke script for a non-Rappture application looks similar to this:

```
#!/bin/sh

/apps/invoke/current/invoke_app "$@" \\
    -t calc \\
    -C calc \\
    -c filexfer \\
    -w captive
```

In the invoke script above, invoke_app, located in the directory /apps/invoke/current, is called with "$@", "-t calc", "-C calc", "-c filexfer", "-w captive". "$@" represents all options that the invoke script itself received. "-t calc" tells invoke_app that the toolname is "calc". This information is used by invoke_app to figure out which tool it is supposed to be launching and where that tool is installed. "-C calc" tells invoke_app that the command to run to start the tool is "calc". "-c filexfer" tells invoke_app to start up the filexfer program before starting the tool's graphical user interface. "-w captive" tells invoke_app to use the icewm captive window manager. For non-rappture applications the icewm captive window manager may be preferred over the ratpoison window manager if there are multiple graphical user interface windows that could popup.

The invoke script above could be made more svelte if the we did not want to start filexfer and we wanted to use the ratpoison window manager. After all, not all applications require files from the user, so they don't need the filexfer program. Here's an example of the tool named calc (the "-t calc" option), that is started by the executable named calc (the "-T calc" option), and uses the default window manager which is ratpoison.

```
#!/bin/sh

/apps/invoke/current/invoke_app "$@" \\
    -t calc \\
    -C calc
```

Other invoke script examples

Here are a few common invoke scripts examples that demonstrate using invoke_app options.

Use the -u option to setup Octave-3.2.4 in the path before starting the tool's graphical user interface. The -u option sources a "use" script (octave-3.2.4 in this example) from the /apps/environ directory.

```
#!/bin/sh

/apps/invoke/current/invoke_app "$@" \\
    -t calc \\
    -C calc \\
    -u octave-3.2.4
```

Use the -A option to send additional arguments to the command to be executed:

```
#!/bin/sh

/apps/invoke/current/invoke_app "$@" \\
    -t calc \\
    -C calc \\
    -A "-value 13 -value 5 -op add"
```

Or:

```
#!/bin/sh

/apps/invoke/current/invoke_app "$@" \\
    -t calc \\
    -C "calc -value 13 -value 5 -op add"
```

Launching a Matlab tool (named app-fermi) with a Rappture graphical user interface:

```
#!/bin/sh

/apps/invoke/current/invoke_app "$@" \\
    -t app-fermi
```

Launching a Python tool (named app-fermi) with a Rappture graphical user interface:

```
#!/bin/sh

/apps/invoke/current/invoke_app "$@" \\
    -t app-fermi
```

Launching a Java tool (named app-fermi) with a Rappture graphical user interface:

```
#!/bin/sh
```

```
/apps/invoke/current/invoke_app "$@" \\
    -t app-fermi
```

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 Outside Computing Resources

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 run dissemination. A set of steps are executed for each run 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 run are gathered and transferred to the remote site. Transferred files include the wrapper and batch description scripts.
- The wrapper script is executed remotely.
- Progress of the remote run is monitored until completion.
- Output files from the run 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:
  -h, --help            Report command usage. Optionally request listi
ng of
                        managers, tools, or venues.
  -l, --local           Execute command locally
  -v, --venue           Remote job destination
  -i, --inputfile       Input file
  -p, --parameters      Parameter sweep variables. See examples.
  -d, --data            Parametric variable data - csv format
  -s SEPARATOR, --separator=SEPARATOR
                        Parameter sweep variable list separator
  -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
  -r NREDUNDANT, --redundancy=NREDUNDANT
```

```
                        Number of indentical simulations to execute in
                        parallel
  -M, --metrics         Report resource usage on exit
  -W, --wait            Wait for reduced job load before submission
  -Q, --quota           Enforce local user quota on remote execution h
ost
  -q, --noquota         Do not enforce local user quota on remote exec
ution
                        host

Parameter examples:

submit -p @@cap=10pf,100pf,1uf sim.exe @:indeck

        Submit 3 jobs. The @:indeck means "use the file indeck as a te
mplate
file." Substitute the values 10pf, 100pf, and 1uf in place of @@cap wi
thin the
file. Send off one job for each of the values and bring back the resul
ts.

submit -p @@vth=0:0.2:5 -p @@cap=10pf,100pf,1uf sim.exe @:indeck

        Submit 78 jobs. The parameter @@vth goes from 0 to 5 in steps
of 0.2,
so there are 26 values for @@vth. For each of those values, the parame
ter
@@cap changes from 10pf to 100pf to 1uf. 26 x 3 = 78 jobs total. Again
@:indeck is treated as a template, and the values are substituted in p
lace of
@@vth and @@cap in that file.

submit -p params sim.exe @:indeck

        In this case, parameter definitions are taken from the file na
med
params instead of the command line. The file might have the following
contents:

                # paramters for my job submission
                parameter @@vth=0:0.2:5
                parameter @@cap = 10pf,100pf,1uf

submit -p "params;@@num=1-10;@@color=blue" job.sh @:job.data

        For someone who loves syntax and complexity... The semicolon s
```

```
eparates
the parameters value into three parts. The first says to load paramete
rs from
a file params. The next part says add an additional parameter @@num th
at goes
from 1 to 10. The last part says add an additional parameter @@color w
ith a
single value blue. The parameters @@num and @@color cannot override an
ything
defined within params; they must be new parameter names.

submit -d input.csv sim.exe @:indeck

        Takes parameters from the data file input.csv, which must be i
n comma-
separated value format. The first line of this file may contain a seri
es of
@@param names for each of the columns. If it doesn't, then the columns
 are
assumed to be called @@1, @@2, @@3, etc. Each of the remaining lines
represents a set of parameter values for one job; if there are 100 suc
h lines,
there will be 100 jobs. For example, the file input.csv might look lik
e this:

                @@vth, @@cap
                1.1, 1pf
                2.2, 1pf
                1.1, 10pf
                2.2, 10pf

        Parameters are substituted as before into template files such
as
@:indeck.

submit -d input.csv -p "@@doping=1e15-1e17 in 30 log" sim.exe @:infile

        Takes parameters from the data file input.csv, but also adds a
nother
parameter @@doping which goes from 1e15 to 1e17 in 30 points on a log
scale.
For each of these points, all values in the data file will be executed
. If the
data file specifies 50 jobs, then this command would run 30 x 50 = 150
0 jobs.
```

```
submit -d input.csv -i @:extra/data.txt sim.exe @:indeck

        In addition to the template indeck file, send along another fi
le
extra/data.txt with each job, and treat it as a template too.

submit -s / -p @@address=23 Main St.,Hometown,Indiana/42
Broadway,Hometown,Indiana -s , -p @@color=red,green,blue job.sh @:job.
data

        Change the separator to slash when defining the addresses, the
n change
back to comma for the @@color parameter and any remaining arguments. W
e
shouldn't have to change the separator often, but it might come in han
dy if
the value strings themselves have commas.

submit @@num=1:1000 sim.exe input@@num

        Submit jobs 1,2,3,...,1000. Parameter names such as @@num are
recognized not only in template files, but also for arguments on the c
ommand
line. In this case, the numbers 1,2,3,...,1000 are substituted into th
e file
name, so the various jobs take their input from "input1", "input2", ..
.,
"input1000".

submit @@file=glob:indeck* sim.exe @:file

        Look for files matching indeck* and use the list of names as t
he
parameter @@file. Those values could be substituted into other templat
e files,
or used on the command line as in this example. Suppose the directory
contains
files indeckA, indeckB, and indeck-123. This example would launch thre
e jobs
using each of those files as input for the job.
```

Additional information is available by requesting user specific lists of choices for some command options. The available option lists are generated for a user based on configured restrictions and availability. The values listed here are for example only and may not be available on all HUBs.

```
$ submit --help tools

Currently available TOOLs are:
   pegasus-plan

$ submit --help venues

Currently available VENUES are:
   DiaGrid
   WF-DiaGrid

$ submit --help managers

Currently available MANAGERs are:
   mpi
   mpich
   parallel
```

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 runs 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 batch submission mechanisms available directly on the submit host. These include PBS, condor, and Pegasus batch queue submission.
- ssh - direct use of ssh. Submit manages access to a common ssh key, essentially serving as a proxy for the HUB user.
- ssh + remote batch submission - use ssh to do batch run submission remotely. Again methods for PBS, condor, and Pegasus batch queue submission are provided.

In addition to single site submission the -r/--redundancy option provides the option to simultaneously submit runs to multiple remote venues. In such cases the successful completion of a run at one venue cancels runs at all other venues. If none of the runs are successful results from one of the runs are returned to the user. Redundant submission is not allowed when performing parametric sweeps.

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

- Execute the command within the user tool session, -l/--local option
- User specified on the command line with -v/--venue option.
- Randomly selected from remote sites associated pre-staged application.
- Select randomly from all configured sites

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 submitted to a batch queue. The job is subject to all remote queuing restrictions and idiosyncrasies.

Remote batch jobs are monitored for progress. Methods appropriate to the batch queuing system are used to check job status at a configurable frequency. A typical frequency is on the order one minute. Job status changes are reported to the user. The maximum time between reports to the user is set on the order of five minutes even in the absence of change. 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 run 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 RUNID.stdout and RUNID.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. RUNID represents unique job identifier assigned by submit.

Pegasus Workflow Submission

Overview

With this version of submit new functionality has been introduced to support workflow management using Pegasus. Two use cases are available: automatic workflow generation for parametric sweeps on one or more variables, or user constructed workflows. In both instances submit is used to configure access to one or more computational resources eliminating the need for a user to supply a site catalog thereby simplifying use of the workflow management system.

Parametric Sweeps

submit command options -p/--parameters and -d/--data have added to provide support for specifying parameter sweeps in a compact general way. The user is relieved of the chore of generating entire sets of input files and command arguments comprising a parameter sweep. Substitutable parameters are declared on the submit command line. Values of these parameters can then be systematically substituted into data files or application command line parameters. submit performs the necessary substitutions to cover all parameter combination. Each combination of parameters is abstractly represented as a node in a workflow and concretely executed as a job on the designated computational resource. A simple curses interface is provided to monitor progress of the simulation run.

User Constructed Workflows

Parameter sweeps are represented as a simple workflow consisting of many individual independent nodes. That is data is not shared between nodes or jobs in the run. There are cases where this simple approach is not sufficient to describe a workflow required to achieve a developer's or user's objective. Under these circumstances a developer may create a workflow and build an application around where the user supplies values for selected inputs. In such cases the Pegasus API's may be used to generate the abstract workflow description in the form of a dax file. The dax file can then executed by a simple submit command.

```
submit pegasus-plan --dax daxFile
```

In cases where more than one venue is capable of executing Pegasus runs a specific venue can be requested on the command line, otherwise submit will choose a venue at random.

```
submit -v DiaGrid pegasus-plan --dax daxFile
```

There are several additional options to pegasus-plan command that are supplied by submit. A few of the command options may be provided on the command line. submit reserves the option to silently ignore options as it sees fit.

In addition to remote execution of Pegasus runs it is also possible to do the execution locally with in the tool session. Simply use the submit -l/--local option.

```
submit --local pegasus-plan --dax daxFile
```

The use command can be employed to put pegasus-plan and all other Pegasus commands in the PATH environment variable. In additional to setting PATH, other environment variables are set allowing use of the python and java dax generation API's.

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.

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

A second code segment is used to build an executable script that can be executed using RapptureExec. 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 and returning control to the application wrapper script.

```
   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`
}

"""
   submitScript += "cd %s\\\n" % (os.getcwd())
   submitScript += "submit -v %s -n %s -w %s \\\\\\n" % (venue,nodes,w
alltime)
   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 = RapptureExec([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.

```
   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 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.

Passing parameters to tools

Overview

This document proposes an updated method of passing parameters to tools when invoking them. We used to do this another way. The intent of this proposal is to make it work properly across all tools on all hubs.

Steps

Step 1: A tool is launched with various parameters as an argument

a) Parameters are defined with a simple syntax that captures the parameter type, an optional parameter name, and its value:

```
directory(context):/tmp
file(input):/data/groups/testgroup/dropbox/input.txt
file:/home/neeshub/mmclennan/.bashrc
```

Each parameter value has the form type(name):value, where type is either file or directory. We may add other types as time goes on, but weâ€™ll never add something general like string. Itâ€™s far too dangerous to let the application parse untrusted string values, so all types must be well-defined and validated by the middleware. b) The tool invocation URL takes a params field value with a newline-separated list of parameters. For example:

```
https://nees.org/tools/indeed/invoke/current?params=directory%28contex
t%29%3a%2ftmp%0a%0dfile%28input%29%3a%2fdata%2fgroups%2ftestgroup%2fdr
opbox%2finput.txt
```

As you can see, the ()â€˜s, :â€˜s, and other punctuation characters are encoded to the URL query notation. But the original text before the encoding for this example was quite simple:

```
directory(context):/tmp
file(input):/data/groups/testgroup/dropbox/input.txt
```

Just like the original example, but only the first two parameters. Note the separator characters %0a%0d in the URL. These are good, since theyâ€™ll never conflict with other syntax and they mimic the syntax that we eventually get in the parameter file.

Step 2: The web server receives and validates the information

The web server examines the params field and parses the syntax for all elements. It scans through and validates all elements, checking their type and value. For file and directory types, the given file path must reside in a white-listed set of known places. For NEES.org, the whitelist will include the /home and /nees directories. The web server will also perform small translations on the params string, like collapsing and stripping extra newlines. If a value is bad, or an argument type is not known, the web server should halt processing and display an error. The web server will not check for the existence of the file or directory because it does not have complete access to the directories and the intended use of the parameter is unknown. Parameters could specify output file names, in which case the file would not exist. File validation is is the responsibilitiy of the application.

Step 3: The middleware is told to start a session with the arguments

The middleware already has an option for appopts. Weâ€™ll leave that alone for backward compatibility and create a similar params option. The params option will receive the URL-encoded params string from the web server. It will decode it using â€˜urllib2.unquote(params).decode(â€œutf8â€•)â€™ and write it to a file called parameters.hz in the userâ€™s session directory. It will also set an environment variable TOOL_PARAMETERS=parameters.hz within the session indicating to the tool that parameter file exists. Thatâ€™s all the middleware needs to do. No need to pass any parameters into the invoke script. The tool (or perhaps the invoke_app wrapper) will take it from there.

Step 4: The tool is invoked.

Many tools will use the new invoke_app script to look for arguments and pass along appropriate arguments. For those like inDEED that may want to handle the arguments themselves, they can. They would simply look for the $TOOL_PARAMETERS environment variable. If set, it points to a file containing the sanitized tool arguments in the form shown earlier:

```
directory(context):/tmp
file(input):/data/groups/testgroup/dropbox/input.txt
file:/home/neeshub/mmclennan/.bashrc
```

The tool would then read this file and parse the various lines to extract arguments.

File Format

Hereâ€™s a proposal for the format of the parameter file. To start with, only 2 types of arguments will be recognized and allowed:

- file parameters (file names)
- directory parameters (directory names)

In the future, we might add others such as url or datastore. A sample file might look something like this:

```
file:/data/groups/me581/assmt1.txt
```

file(diagram):/home/nanohub/mmc/indeed/cache0012
directory(context):/tmp

In general, the syntax is

```
type:value or type(name):value.
```

The value for the file keyword must be an absolute file path for the desired file. The cms and middleware do not perform file path existence checks. The cms checks the for an allowed type and will only check that the url encoded string matches a set of The value for the directory keyword must be an absolute directory path. The middleware checks the file path to make sure that the directory exists and throws an error if not. If we added a datastore type some day, the value might be some sort of table identifier, optional view identifier, and an optional series of rows. For example, a value like

```
datastore:132/stdview/1,5,8
```

where 132 is the datastore table, stdview is the view name, and 1,5,81 are the desired rows. These values are separated by /. Any parameter can have an optional name, so that clients can request values by name via an API call. Parameters can also be referenced by index, such as #1, #2, and so forth. Typical Use Case In a typical use case, a professor wants to put a link to a tool in a wiki page, so that students can easily start the tool with a specified file name. The HTTP GET request for that case might look something like this:

```
http://nanohub.org/tools/octaview/invoke/current?params=file%3a%2fdata
%2fgroups%2fme581%2fassmt1.txt
```

Here, params is the parameter string, and there will be only one line in the parameter specification file with

```
 file:/data/groups/me581/assmt1.txt.
```

If a tool needs to load 2 files, then the params list should include two file declarations separated by the escape sequence %0a%0d (CRLF).

Shell Commands

A typical use case for parameter passing is passing one or more file names as command-line arguments. We want to make that easy for many tools and add appropriate error checking. To do this, we will extend the existing "invoke_app" command to handles the following syntax:

```
invoke_app ... -C  <command> ?-C <command> ...? ?-C <command>?
Tries to find a  template that matches a set of parameters
and executes that command.  Each  can contain embedded
references to parameters:
@@type(name) ... parameter with the specified type and name
@@type(#N) ..... parameter with the specified type and position #N on
the command line
```

The parameter type can be either "file" or "directory". (More types may be added later.)

Examples

```
invoke_app "$@" -C 'firefox @@file(url)' -C 'firefox'
```

If thereâ€™s a file parameter named url, then run firefox with that as an argument. Otherwise, run firefox with no arguments.

```
invoke_app "$@" -C 'tkdiff @@file(#1) @@file(#2)' -C 'gedit @@file(#1)
'
```

If there are two file parameters, then run â€œtkdiffâ€• on the two files. Otherwise, run â€œgeditâ€• with one file. There is no default command (with no parameters), so there must be at least one file parameter or the tool will throw an error.

Errors

If a matching command can be found and there are no other errors, then the command is executed and thatâ€™s that. But if there are no matching commands, then toolparams shows an error dialog in the tool session. If there are subtle errors (such as extra parameters that are not used in the command template), then toolparams pops up a smaller window in the lower-right corner with error messages.

Rappture API

How to pass parameters to Rappture tools

This document proposes a method of passing parameters to tools built with Rappture.

Small step: Automatically use parameters to set the default values for string Rappture elements.

Use the current parameter syntax to describe the Rappture element to be set. Example:

```
file(input.string(indeck).default):/path/to/file.txt
```

Rappture reads in the parameters.hz file if one exists and sets up an associative array of element paths. The value of each path is the name of a file. In the previous example, the textentry widget in Rappture will check to see if it has a parameter entry for itself (.e.g $path.default) in the global parameters array. If it does, it will open the file and use the contents of the file as the default value (ignoring the specified default value in the tool.xml). If you use XML paths in parameters, it makes your the tool fragile. If you change your tool.xml, you have to fix all the URLs. One possible remedy is to create a mapping in the tool.xml. This maps a name to an XML path.

```
<parameters>
<parameter>
<name>indeck</name>
<variable>input.string(indeck).default</variable>
</paramater>
</parameters>
```

Then the parameter would be

```
file(indeck):/path/to/file.txt
```

On the other hand, specifying XML paths has the advantage of setting parameters on any Rappture tool on any XML tag, not just ones that the tool author specified by a parameter mapping.

Bigger step: Extend the parameter passing syntax to include â€œstringâ€œ parameters.

This will let you inline the values in the URL. Example:

```
string(input.number(temperature).default):300K
```
string(input.choice(model).default):boltzman

The real advantage of the inline string is you can then override any tag in Rappture.

```
number(input.number(temperature).min):3K
```
number(input.number(temperature).max):10000K

Loader issues

The parameter passing mechanism has to cooperate with the loader. You could override any XML value with a parameter on construction. This makes sure you load the parameters only once. The loader overwrites the parameter value with itâ€™s default loader value. This happens after the construction of the widgets. If you set parameters after the widgets initial construction, then you can only set current values, not any tag.