

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 `/usr/bin`. 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 environment packages
-v visualization server version
```

TOOL DEVELOPERS

`-w` specify alternate window manager

Here is a detailed description of the options:

<p><code>-A</code></p>	<p>pass the provided enquoted arguments onto the tool.</p> <p>Example usage:</p> <pre>-A "-q blah1 -w blah2"</pre> <p>The options <code>-q</code> and <code>-w</code> are not parsed by <code>invoke</code>, but are passed on to the tool</p>
<p><code>-c</code></p>	<p>Commands to run in the background before the tool launches.</p> <p>Exmple usage:</p> <pre>-c "echo hi" -c "filexfer"</pre> <p>This prints "hi" to stdout and starts <code>filexfer</code></p>
<p><code>-C</code></p>	<p>Command to execute for starting tool. Tool's command line arguments can be included in this option, or can be placed in the <code>-A</code> option.</p> <p>Example usage:</p> <p>Call a program, named <code>myprog</code>, located in the tool's bin directory:</p> <pre>-C @tool/bin/myprog</pre> <p>Call a program, named <code>myprog</code>, located in the tool's bin directory, with program arguments <code>"-e val1"</code> and <code>"-b val2"</code>:</p> <pre>-C "@tool/bin/myprog -e val1 -b val2"</pre>

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:

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

TOOL DEVELOPERS

	<p>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.</p> <p>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.</p>
-t	<p>sets <code>{toolname}</code> which is used while setting up tool paths for TOOLDIR and TOOLXML. <code>{toolname}</code> 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 <code>/apps/toolname/version/rappture/tool.xml</code>. Setting this option will change the paths searched while trying to locate tool.xml and the bin directory.</p>
-T	<p>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 <code>/apps/toolname/version</code> in the path <code>/apps/toolname/version/rappture/tool.xml</code>. 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:</p> <pre>-T \$PWD</pre>
-u	<p>Set use scripts to invoke before running the tool.</p> <p>Example usage:</p> <pre>-u octave-3.2.4 -u petsc-3.1-real-gnu</pre> <p>These would setup octave-3.2.4 and petsc-3.1 in the environment that your tool would launch in.</p>

<p>-v</p>	<p>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.</p> <p>Example:</p> <pre>-v dev</pre> <p>This option irrelevant if no visualization server is available.</p>
<p>-w</p>	<p>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"</p> <p>Examples:</p> <p>Use the icewm captive window manager.</p> <pre>-w captive</pre> <p>Use the ratpoison window manager.</p> <pre>-w ratpoison</pre>

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
```

```
/usr/bin/invoke_app "$@" \\  
\\
```

```
-t calc
```

In the invoke script above, `invoke_app`, located in the directory `/usr/bin`, 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:

```
/usr/bin/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

/usr/bin/invoke_app "$@" \\  
    -t calc \\  
    -C calc \\  
    -c filexfer \\  
    -w captive
```

In the invoke script above, `invoke_app`, located in the directory `/usr/bin`, 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-rapture 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

/usr/bin/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

/usr/bin/invoke_app "$@" \\  
    -t calc \\  
    -C calc \\  
    -u octave-3.2.4
```

TOOL DEVELOPERS

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

```
#!/bin/sh

/usr/bin/invoke_app "$@" \\  
    -t calc \\  
    -C calc \\  
    -A "-value 13 -value 5 -op add"
```

Or:

```
#!/bin/sh

/usr/bin/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

/usr/bin/invoke_app "$@" \\  
    -t app-fermi
```

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

```
#!/bin/sh

/usr/bin/invoke_app "$@" \\  
    -t app-fermi
```

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

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

TOOL DEVELOPERS

```
/usr/bin/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
```


TOOL DEVELOPERS

```
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.

TOOL DEVELOPERS

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 wrapper script, selected description scripts.
- 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 listing 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
```

TOOL DEVELOPERS

	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 host
-q, --noquota	Do not enforce local user quota on remote execution host

Parameter examples:

```
submit -p @@cap=10pf,100pf,luf sim.exe @:indeck
```

Submit 3 jobs. The @:indeck means "use the file indeck as a template file." Substitute the values 10pf, 100pf, and luf in place of @@cap within the file. Send off one job for each of the values and bring back the results.

```
submit -p @@vth=0:0.2:5 -p @@cap=10pf,100pf,luf 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 parameter @@cap changes from 10pf to 100pf to luf. 26 x 3 = 78 jobs total. Again @:indeck is treated as a template, and the values are substituted in place of @@vth and @@cap in that file.

```
submit -p params sim.exe @:indeck
```

In this case, parameter definitions are taken from the file named 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,luf
```

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

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

TOOL DEVELOPERS

eparates

the parameters value into three parts. The first says to load parameters from

a file params. The next part says add an additional parameter @@num that goes

from 1 to 10. The last part says add an additional parameter @@color with a

single value blue. The parameters @@num and @@color cannot override anything

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 in comma-

separated value format. The first line of this file may contain a series 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 such lines,

there will be 100 jobs. For example, the file input.csv might look like 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 another

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 = 1500 jobs.

TOOL DEVELOPERS

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

In addition to the template indeck file, send along another file 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, then change back to comma for the @@color parameter and any remaining arguments. We shouldn't have to change the separator often, but it might come in handy 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 command line. In this case, the numbers 1,2,3,...,1000 are substituted into the 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 the parameter @@file. Those values could be substituted into other template 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 three jobs using each of those files as input for the job.

Additional information is available by requesting user specific lists of choices for some

TOOL DEVELOPERS

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

TOOL DEVELOPERS

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
```

TOOL DEVELOPERS

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 addition 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 be 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"
```

TOOL DEVELOPERS

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

append submitScript "cd [pwd]\n"
append submitScript "submit -v cluster -n $nodes -w $walltime\\\\\\\\\n"
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
```

TOOL DEVELOPERS

```
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
_IROTH)
```

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()
stderr = ''.join(outFileLines)
out2 += '\n' + stderr
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()
            stdoutoutput = ''.join(outFileLines)
            out2 += '\n' + stdoutoutput
            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 be 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);
}
```

TOOL DEVELOPERS

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.

Accessing your home directory

Overview

Accessing your home directory on the HUB is easy with the three methods described in this section. While sFTP is the most common, you will find which method works best for you.

sFTP

Accessing your home directory via sFTP

sFTP, or secure FTP, is a program that uses SSH to transfer files. Unlike standard FTP, it encrypts both commands and data, preventing passwords and sensitive information from being transmitted in the clear over the network. It is functionally similar to FTP, but because it uses a different protocol, you can't use a standard FTP client to talk to an sFTP server, nor can you connect to an FTP server with a client that supports only sFTP.

The following tutorial should help you in using sFTP to connect to and from your HUBzero server(s).

Warning: Most accounts do **not** have SSH/sFTP access initially. Your system administrator must grant your account access before you will be able to connect.

Graphical Clients

Using graphical SFTP clients simplifies file transfers by allowing you to transmit files simply by dragging and dropping icons between windows. When you open the program, you will have to enter the name of the host (e.g., yourhub.org) and your HUB username and password.

Windows Clients

- [WinSCP](#)
- [BitKinex](#)
- [FileZilla](#)
- [PuTTY](#)

Mac OSX Clients

- [Transmit](#)
- [Fetch](#)
- [Cyberduck](#)
- [Flow](#)
- [Fugu](#)

Command-line

You can use command line SFTP from your Unix account, or from your Mac OS X or Unix workstation. To start an SFTP session, at the command prompt, enter:

```
yourmachine:~ you$ sftp username@host
yourmachine:~ you$ username@host password:
```

```
host ~
```

TOOL DEVELOPERS

Some standard commands for command-line sFTP	Command	Description
	cd	Change the current directory
	chmod	Change the permissions of a file
	chown	Change the owner of a file
	dir (or ls)	List the contents of the current directory
	exit (or quit)	Close the sFTP session and return to the local shell
	get	Copy a file from the remote system to the local system
	help (or ?)	Get help for a command
	lcd	Change the local directory
	lls	See the local directory listing
	ln	Create a hard link
	ln (or symlink)	Create a symbolic link
	lpwd	Show the local path
	lumask	Change the local umask
	mkdir	Create a directory
	put	Copy a file from the local system to the remote system
	pwd	Show the remote path
	rename	Rename a file
	rm	Delete a file
	rmdir	Remove a directory
	version	Display the sFTP version
	!	In Unix, !pwd drops the user to the local shell

WebDAV

Accessing your home directory via WebDAV

WebDAV is the Distributed Authoring and Versioning extension to the standard HTTP/HTTPS web protocol. It allows a client to browse a remote filesystem, usually with a graphical browser that makes it appear that your files are on your desktop. You may access your hub storage using only the secure version of this service (HTTPS). We do not support HTTP. Most modern computer platforms support HTTPS transport for WebDAV with either small adjustments or freely available software.

Linux/Unix

If you use the KDE graphical desktop environment, you can access your hub storage with the Konqueror browser by typing the special URL `webdav://hubzero.org/webdav` into the Location field of the browser. It will prompt you for your hub login and password. Thereafter, you traverse your home directory by clicking on folders and you can drag and drop files to your desktop.

[Cadaver](#) is a text-mode WebDAV browser. It can be used if it is compiled with SSL support. Invoke it with the command `cadaver https://hubzero.org/webdav/` and it will prompt you for your hub login and password. You can then use it in a manner similar to FTP.

If you are using Linux, you can use the [davfs](#) kernel module to mount your hub storage area as a local filesystem.

Macintosh

MacOS versions 10.4 and higher support HTTPS transport for WebDAV using the Finder. To access your hub directory, select the Go menu in the Finder and choose "Connect to Network Server". Enter the URL `https://hubzero.org/webdav/` into the address field. You will be prompted for your hub login and password.

MacOS versions lower than 10.4 do not have support for HTTPS, but a free client exists.

Download [Goliath](#) and install it on your system. Invoke it and use `https://hubzero.org/webdav/` as the address to connect to. Note that you may also use Goliath on newer MacOS platforms as well.

Windows

Windows XP systems have native support for accessing WebDAV storage via HTTPS, but it may be disabled by default depending on how your OS was configured. Try these instructions first, and then check the next set of instructions to enable WebDAV. To access your hub storage,

- Double-click on the desktop icon called "My Network Places".
- This will invoke a dialog box where you can select "Add Network Place". This will invoke a wizard that you can step through.

- Select the "Choose another network location" function.
- The wizard will then ask you for the "Internet or Network Address." In the box, type `https://hubzero.org/webdav/`
- Specify the name that you would like to have associated with this storage space. An icon will be created for the hub storage folder. When you double-click on it to open it, you will be prompted for your hub login and password.

Enabling WebDAV in Windows XP

If the steps above do not work, for Windows XP, or you cannot find the named options, your OS may have WebDAV turned off by default. Do the following additional steps and then try to add network storage again.

- Open Windows Explorer
- Go to the "Tools" menu and select "Folder Options"
- Select the radio button "Use Windows classic folders" under "Tasks".

Windows 2003 users cannot access secure WebDAV

There is no support for secure WebDAV transport in Windows 2003. Various workarounds have been suggested, but it appears the easiest action to take is to simply upgrade the OS.

Windows Vista users cannot access secure WebDAV

A default and up to date installation of Windows Vista has no support for secure WebDAV. You'll need to install the following [patch](#).

Windows XP users cannot change credentials for WebDAV authentication.

To clear previously entered authentication data from the credential cache, use the following menu selection:

Start -> Settings -> Control Panel -> User Accounts -> Change an Account

Then select the User account for which you wish to clear the stored password. Click on 'Manage My Network Passwords', highlight the mapped drive or WebDAV account and click 'Remove'.

Windows users trick

Many users have mentioned that changing a certain setting in their internet options has allowed them to access a site via WebDAV (this includes several instances on Windows 7). Open up the "Internet Options" from the control panel, go the "Connections" tab, click on the "LAN Settings" button, and insure that "Automatically detect settings" is unchecked.

This does not guarantee that WebDAV will work in every instance, but has simply proven to be a helpful trick for many users.

filexfer (in Workspace tool)

Accessing your home directory files via filexfer (in Workspace tool)

Filexfer, short for 'file transfer', is a utility that you can call from within the workspace tool to transfer a file into your home directory from your local machine or download a file from your home directory. Type 'filexfer' in the xterm provided to you when you start the workspace tool. You should now see the filexfer GUI as it appears in the image below.

Click on "Upload" or "Download" for the desired action. Please note, that you must enable pop-ups in your browser to proceed. The following windows that appear allow you to browse or download per your selected action. Note: filexfer is currently limited to uploading one file at a time..