

## Resource Tuning Session

Session between Parag and Angela, Monday, July 2, 2007. Command line entries are in **bold Courier New font** and begin with a dollar sign (\$). Code, products, executables, etc. are in plain Courier New font. **Highlighted portions** have a pop-up note embedded with them because I had some question about it.

### General information

There are three types of data queues:

- **Application**
  - example: `binary_fetch`
  - used to move executables around
- **Executable**
  - example: `dzero_reco`
  -
- **stager**
  - example: `??`
  - prepares files to be moved to/from a more durable location

People who will be doing any modifications should be added to the `.k5login` file for user "samgrid".

### Set up

We want to make some modifications to the resource tuning configurations. First we log in to **appropriate node**, in this case `d0srv047`. Preferably log in as user "samgrid" to do this stuff, but one could log in with a root principle. Angela could not get her root principle to work, so Parag logged in with his root principle. To get the general environment set up, type:

```
$ source ~sam/setup.sh
```

We know we will need to modify configuration files for the product "jim\_job\_managers", so set that up:

```
$ setup jim_job_managers
```

To modify things, we will be using a script called "jim\_configure.sh". In order to put that script in our environment path, we do:

```
$ setup samgrid_util
```

### Looking at an existing example

#### *Overall queue configuration*

All of the configuration files used here are written in XML code. If we want to see a list of the existing

queues on this machine, we can look at the main queue configuration file using:

```
$ jim_configure.sh jim_job_managers
```

This opens a configuration file for `jim_job_managers` using the “vi” editor (this editor choice is hard-coded in the script). Let's look at an example input queue called “`dzero_reconstruction`”:

```
<dzero_reconstruction>
  <input_storage name="raw">
    <prot_fcp queueName="raw_download"/>
```

The `<input_storage>` tag attribute “name” does not have any formatting restrictions. It just has to be unique per queue configuration. For the tag called “`<prot_fcp . . >`”, the “prot” is short for “protocol” and “fcp” is a file transferring product so the attribute “queueName” of this tag describes the transfer method to use.

 Note: Executables are broken down by type into queues, and then further broken down by the type of connection (“local,” “lan,” or “wan”).

### **`<Input_storage>` configuration**

The `<input_storage name="raw">` tag needs its own definition, which is done in the `jim_config` configuration file. To edit the file, do:

```
$ jim_configure.sh jim_config
```

In this case `<input_storage name="raw">` has already been configured. The entry looks like this:

```
<input_storage name="raw" location_selector_algorithm="random"
location_selector_pattern="d0sam01.fnal.gov.*cache1.*| etc.>
```

The “`location_selector_algorithm`” attribute must be either “random” or “affinity”. For the “random” type, the “`location_selector_pattern`” is a list of regular expressions<sup>1</sup> of the form `domainname::queuename` with each item in the list separated by a single logical OR (represented by a single pipe character “|”). The domain-queue combination used is selected randomly from the expressions in the list. The “affinity” type has a similar “`location_selector_pattern`” except that the regular expression list is separated by a double logical OR (represented by double pipes “||”). The domain-queue combination to use is selected by going through the items in the list in order and looking for a match. In some cases an “affinity” type will have a pattern list that contains both single ORs and double ORs. Such a list is parsed by splitting at the double ORs first, with any entries containing single ORs combined into a separate item list. The non-list items are considered first for matches. If no match is found, then the entries in the single-OR list are considered as if they were a “random” pattern list. 

---

<sup>1</sup> Most of these expressions are matched character-by-character with the star (“\*”) being a wildcard for any number of characters.

## <Prot\_fcp> configuration

The <prot\_fcp> tag needs to be defined as well. This definition is in the sam\_fcp configuration file. First, set up the package, and then edit the configuration file:

```
$ setup sam_fcp
$ jim_configure.sh sam_fcp
```

Again, this tag has already been defined so no modifications were needed. The definition includes the maximum number of transfers allowed, the time-out value, the transfer mechanism, etc.

## Aside - Running jobs

To see what jobs that are running on the node you are logged in to, use:

```
$ ps -efwww | grep jobmanager
```

## Configuring <dzero\_monte\_carlo>

First, we'll edit the jim\_job\_managers configuration file. A partial entry had already been created, so we just scrolled to it. We will configure the “executable” first. There are two choices: “binary” or “raw”. “Binary” is used to move the actual binary executable file(s) to its running location. “Raw” is for moving input files that would be used by an executable. Since the executable does not currently consume input files, we will only need a “binary” entry. A set of 'empty' <local\_data\_buffer> tags was added so that the entry now looks like this:

```
<dzero_monte_carlo>
  <input?.....>
    <local_data_buffer>
  </local_data_buffer>
</dzero_monte_carlo>
```

The empty tag set will pick up the 'default' of “<binary\_fetch>”, which is the protocol used to transfer executable files. There is currently no way to explicitly designate binary\_fetch in the main configuration file. The <binary\_fetch> tag has already been used in other configurations, and so we do not need to define it ourselves. The definition includes the directive to use the “rte” file queue. If we had needed to configure binary\_fetch, we would do that in the “jim\_config” configuration file. To edit the file, we would have used:

```
$ jim_configure.sh jim_config
```

We can run a test on the executable configuration to see which queue actually gets selected with each call. The command for the test is of the form:

```
$ jim_sam_storage_negotiator_cmd.py <direction> <stationname>
<applicationname>
```

where <direction> is either the key word “store” or the key word “retrieve”, <stationname> is the name of the station you are trying to negotiate with, and <applicationname> is the name of the executable you are trying to “run.” The example we used was:

```
$ jim_sam_storage_negotiator_cmd.py retrieve osg-ouhep binary_fetch
```

The output of the command looks like this:

```
Std Err:  
Std Err:  
LOCATION=<nodename>  
ACCESS_Q=<queueName>
```

The “StdErr:” lines were blank for our example., The *<nodename>* is the  (?) to the executable to be used, and *<queueName>* is the queue configuration that would be used.

The next step would be to configure input files, but the Monte Carlo executable does not currently consume input files. Whatever calibration or input files it needs it grabs during execution.

The last step is to set up the output configuration. Unfortunately Parag is unsure what resources are to be used for the output, so we stopped the exercise at this point. When Parag gets the information that is needed, we will get together again and finish the implementation. We have not set a specific time yet. To make sure we didn't cause trouble by leaving a partially-configured entry in the configuration file, we also went back and took out the *<local\_data\_buffer>* tags.