Using PBS Professional Hooks: Examples and Benefits

Martin Nicholls

March 2012

www.pbsworks.com

Abstract

This paper examines practical uses of the PBS Professional® hook mechanism, to help achieve site requirements, based on our experiences at The University of Queensland Research Computing Centre. The paper includes descriptions of several real-world applications for job submission hooks (qsub_hook) and includes example code to help commence implementation of a generic hook able to handle both job submission and job modification events.

Introduction

PBS Professional's version 10 introduced the capability for site customization through its "Hook" feature. The Hook feature allows Python code subroutines to be invoked on the PBS Server in response to the execution of a client command (e.g,.qsub) or other events. Each hook script may be configured to respond to a particular event or set of events. In the examples here we will be looking mainly at the "queue_job" event triggered by the "qsub" command but we will also examine the "modify_job" (qalter) event.

Previously, the University of Queensland (UQ) Research Computing Centre (RCC) site has used a Perl wrapper program in front of the "qsub" command to validate user job requests. The qsub_hook allows for a more robust implementation of this functionality and can also handle job modifications. The site can define a single monolithic qsub_hook script or multiple scripts each handling specific validations.

PBS Professional's qsub_hook is triggered when a user executes the PBS Professional qsub command. PBS Professional runs the site Python code routines on the PBS server as a sub-task of the pbs_server process. An object-based event model is used to communicate the user request parameters to the Python code. The qsub_hook runs after the command and job script file parsing phase, and prior to the server database update. Thus, the user code is relieved of command parsing but may correct or modify the user request prior to its being registered.

The hook script may provide an error message if it rejects the job. When a job is accepted it may still be desirable to notify the user of any job modification, or provide warnings or advice for future jobs. For accepted jobs there is no direct communication to the users terminal, but messages may be delivered indirectly.

Hooks scripts should consume limited time and resources. If execution exceeds a configurable maximum (Alarm) time, execution is terminated and the job submission rejected. Hooks should therefore minimise dependencies and avoid unpredictable delays such as NFS access.

The most common use for hooks is likely to be the validation of client jobs requests. At UQ there are two qsub_hooks that validate particular job resources. A third hook is used to optimize scheduling by setting a number of custom job resources.

Sample Hook Uses

Validation of Custom Resources

UQ has a heterogeneous HPC cluster consisting of four different compute node configurations. Some nodes have been configured with additional memory and disks to assist specific applications. A PBS Professional custom node resource “NodeType” was created to allow clients to request a particular node configuration. This resource is also useful to ensure MPI jobs run across a homogeneous set of nodes.

Initially a large number of clients erred when using this resource. Common problems included a mismatch in the text case of either the resource name or the value, mis-spelled and random NodeType values. These resulted in the jobs failing to run or running without the desired restriction.

To solve this problem, a qsub_hook was written to validate both the case of the resource name and resource values. PBS Professional is now able to automatically correct the text case and can reject job submissions with an unrecognizable NodeType resource value. This allows the client to correct the problem at job submission time and avoid delay.

Job Accounting and Reporting

Reporting on system usage is important at UQ, where multiple stakeholder groups share usage and funding of the cluster. Some individual users work on projects for two or more stakeholder groups; thus the usage must be recorded against a project rather than simply against a user.

PBS Professional allows jobs to be individually assigned an “Account Name” string with the “qsub -A” option. This account name is not used by PBS Professional itself but is made available within job records for clearer system usage accounting.

A qsub_hook is used to set or enforce a mandatory account name on all jobs. Many users have a single project and the hook code can automatically set this project as their Account Name value. For users with multiple projects, the hook code will ensure a valid project account name is specified for the job and ensure that the user is a member of that project group.

At UQ, the user project account membership is maintained within the Unix group database with valid project account names being a subset of group names matching a set of patterns. The hook script must read this account membership data. To minimize the input delay, the data is held on the root disk in a prepared summary file.

Scheduler Assistance

At UQ cluster has a diverse mix of job sizes, run lengths and diverse client requirements, making the process of job scheduling challenging. A qsub_hook was written to set several custom job resources that help optimise scheduling decisions:

- **cpu_hrs** is the product of the number of CPUs requested and the walltime request (see PBS Admin Guide example) that provides a useful comparative measure of the job size. For jobs requesting large memory, a proportionate number of CPUs is estimated for the UQ node hardware.

  The cpu_hrs resource is employed to reduce the queue wait time for larger jobs. At UQ, cpu_hrs is used as the highest weighted term within a custom job scheduling formula. This ensures that in each scheduler cycle, larger jobs are considered first and given resources before any of the available resources are consumed by smaller jobs. This has been successful in reducing the queue wait time for the variety of larger jobs.

This resource as measure of current per user usage has also proven helpful to manage user sharing of the system. It is configured as a server resource limit.

```
set server max_run_res.cpu_hrs =  [u:PBS_GENERIC=43008]
```

- **schedmins** is the job "duration in minutes" taken directly from the job walltime. This resource was created to improve job access to all 8 cpus on a compute node. On nodes running "long" jobs, the node "schedmin" resource is set to the duration until it's longest job completes. That node will not accept jobs that would extend its busy time, but can continue to accept shorter jobs. This guarantees that at the end of any long job the node will become completely idle and all 8 cpus will become available simultaneously.
  This mechanism has been proven effective in improving the wait time of jobs that require all 8 cpus on nodes. The feature also ensures that the nodes become available within a reasonable timeframe for software updates.

- **schedclass** is an experimental job classification designed for scheduler node selection. The aim is to help special (scarce) resources to be quickly available for those jobs that request them, without the resource remaining idle.
  The schedclass identifies jobs that require any special resources and also short jobs that could run on those resources without effecting quality of service.
  While each job is assigned a schedclass, each node has its "schedclass" set to a list of the job classes it will accept.
  The attribute is current used for experimental purposes and not within the production scheduler.

Practical Hook Implementation

Debugging Techniques

A hook must often be tested on a live server. This can be done most safely by limiting the code to a single test account, as shown in the example below:

```
je = pbs.event()
user = je.requestor
if user != “martin”:
    je.accept
```

Debug text and status cannot be output directly on the job submission terminal since the hook runs on the PBS server and not on the client’s system.

Debug messages may be written to the PBS Professional server logs where the developer will normally have read access.

An alternate approach that may also be used for production messages to users is to put short messages into the job comment.

```
jb.comment = "Warning : Job array will run for "+ \
    str(dur_days)+" days."
```

A simple bash wrapper script can be used to copy any job comment to the invoking terminal, as shown below:

```
#!/bin/bash
#
# UQ wrapper for PBSpro qsub
#
# The function of this wrapper is to report any
# "Job comment" to the user.

me=`basename $0`
. /etc/pbs.conf

QSUB="$PBS_EXEC/bin/qsub"
#if [ "$me" == "qsub" ]; then
#   QSUB="$PBS_EXEC/bin/qsub.actual"
#fi

# Must detect and handle interactive jobs
```

```
for opt in "$@"; do
   if [ "--" == "$opt" ]; then break; fi
   if [ "-I" == "$opt" ]; then exec $QSUB "$@"; fi
done
while read line; do
    echo $line
    if [ ${line/*.pbsserver/XXX} == “XXX” ]; then
export jobid=$line; fi
done < <( $QSUB “$@” )
comment=$($PBS_EXEC/bin/qstat –f $jobid|awk \
  ‘{if($2==”=”)p=0;if(p==1)printf
substr($0,2)}/comment =/{p=1;printf substr($0,15)}’)
echo $comment
```

Handling Both New and Old Resource Request Forms

When a hook is triggered by a “queuejob” event it receives an “event object” which points to a new “job object” constructed from the qsub command parameters and the job script. The hook is able to make changes to this prototype object prior to it being added to the server database.

PBS job resource requests may be given as either job-wide resources or within the newer selection string format. A hook script must look for resource values in both forms. Example code is shown below. Simple hooks interested in only one or two specific resources can check for those specifically in the Resource_List, making the code for that format straightforward. For more advanced hooks, the select string format must always be parsed in full. Here resources are automatically discovered and need not be specifically named in the code. It is useful to collect the resources and their values into a Python dictionary for any further processing.

```
Res_list = job.Resource_List
if Res_list[“select”]:
     # Parse and disassemble the select string
     …
else:
    for res in [ “host”, “vnode”, “nodes” ]:
        if Res_list[res]:
            reqs[res] = Res_list[res]
```

Parsing a Select String

When a job has been submitted with a “select” string it is necessary to parse the string to extract each resource value pair. A hook that needs to correct or update the request will need to rebuild a new “select” string including all component chunks. The code may avoid saving individual chunk details by generating the new select string during the parse phase. More typically code will be interested only in the total resource counts.

Below is a simple select string parser.

```
if Res_list[“select”]:
    sel = repr(Res_list["select"])
    for chunk in sel.split("+"):
        if sel2:
            sel2 += "+"
        for rs in chunk.split(":"):
            kv = rs.split("=")
            if len(kv) == 1:
                reqs[“nodes”] = reqs[“nodes”] + kv[0]
                sel2 += kv[0]
            if len(kv) == 2:
                reqs[kv[0]] = kv[1]
```

Handling “modifyjob” Events

Previously examples have examined the “queuejob” event where the job is described completely by the event “job object”.

In a “modifyjob” event, the event “job object” contains only the changes to the job. Job validation will often require testing these new values against job attribute values that are not being changed. The script must find the original values from the server object.

In the example code below the script is testing that the job may be modified before extracting the current job parameters.

```
if je.type == pbs.MODIFYJOB:
    job = pbs.server().job(jb.id)
    if str(job.job_state) == “4”:
        je.accept()
    if job.job_state in [ pbs.JOB_STATE_RUNNING, \
        pbs.JOB_STATE_EXITING, pbs.JOB_STATE_TRANSIT ]:
        je.accept()
    # extract the necessary resources
```

Conclusion

We have shown that PBS Professional hooks may be used to validate user job requests against the local site requirements and available resources. This helps reduce jobs with user errors and improves the overall user experience. We have also seen PBS Professional hooks used to optimize specific site scheduling issues. PBS Professional hooks provide a powerful and efficient mechanism for a site to implement custom changes to the PBS Professional commands.

Appendix: Expanded Code Example

```
# PBS Hook - Validate Job Request
#
#  Written : Martin Nicholls
#
import pbs
import sys
import re
import math

## Interpret job request values
def rd_resource( reqs, res, val ):
    # Read a resource value and save in "reqs
    # If we find the value already set, don't overwrite it
    global je, New_val, chunksize, totnodes, totcpus, ntvalues
    if not res or not val or res in reqs:
        return
    # Validate and correct text case on NodeType parameter
    if res.lower() == "nodetype":
        if not val.lower() in ntvalues:
            values = ""
            for v in ntvalues:
                values = values+" "+v
            je.reject("Invalid NodeType value "+val+
                ",Select from :"+values+". ")
        if res != "NodeType" or not val in ntvalues:
            res = "NodeType"
            val = val.lower()
            New_val[res] = val
        if val == "any":
            val = ""
            New_val[res] = "None"
        else:
            reqs[res] = val
    elif res == "mem" or res == "vmem" or res == "scratch":
        reqs[res] = pbs.size(val)
    elif res == "ncpus":
        reqs[res] = int(val)
        totcpus += int(val) * chunksize
    elif res == "walltime":
        reqs[res] = int(val)
    else:
        reqs[res] = val
        if res == "nodes":
            chunksize = int(val)
            totnodes += int(val) * chunksize
    return
# End of rd_resource

def read_req( job, reqs ):
    # Read job resource requests, from either a select statement
or directly in Resource_List (old style)
    Res_list = job.Resource_List
    if job.interactive:
        rd_resource( "interactive", "true", reqs )
    if Res_list["select"]:
        # Look for resources within select statement
        sel = repr(Res_list["select"])
        nodes = 1
        nchunks = 0
        for chunk in sel.split("+"):
            nchunks += 1
            for rs in chunk.split(":"):
                kv = rs.split("=")
                if len(kv) == 1:
                    rd_resource( reqs, "nodes", kv[0] )
                if len(kv) == 2:
                    rd_resource( reqs, kv[0], kv[1] )
    else:
        # Look for old style resource requests
        for res in 'nodetype' 'Nodetype' 'nodeType':
            if res in Res_list:
                rd_resource( reqs, 'NodeType', Res_list[res] )
        for res in requestable:
            if res in Res_list:
                rd_resource( reqs, res, Res_list[res] )

    return
# End of read_req

## MAIN Program
try:

    je = pbs.event()
    jb = je.job

    # For development and testing
    user = je.requestor
    if user != "martin":
        je.accept()

    reqs = dict()
    New_val = dict()
    chunksize = 1
    totnodes = 0
    totcpus = 0

    # Look for resources
    select = read_req( jb, reqs )

    # Handle modify job events by checking existing values too
    if je.type == pbs.MODIFYJOB:
        job = pbs.server().job(jb.id)
        if str(job.job_state) == "4":
            je.accept()
        if job.job_state in [ pbs.JOB_STATE_RUNNING,
            pbs.JOB_STATE_EXITING, pbs.JOB_STATE_TRANSIT ]:
            je.accept()
        select2 = read_req( job, reqs )

    # Set walltime (as this may be outside 'select' clause)
    if "walltime" in jb.Resource_List:
        reqs['walltime'] = int(jb.Resource_List["walltime"])
    elif job and "walltime" in job.Resource_List:
        reqs['walltime'] = int(job.Resource_List["walltime"])
    else:
        # No walltime given
        je.reject("Walltime must be specified, jobs can not ”+ \
            “be scheduled without a runtime estimate!")

# If any values have changed we need to update
# the "select" string and job
    if len(New_val) > 0:
        write_req( jb, New_val )

    je.accept()

except SystemExit:
    pass

except:
    je.reject("%s hook failed with %s, Please contact Admin"
        % (je.hook_name, sys.exc_info()[:2]))
```