Submit

Introduction

The submit command provides a means for HUB end users to execute applications on remote resources. The end user is not required to have knowledge of remote job submission mechanics. Jobs can be submitted to traditional queued batch systems including PBS and Condor.

Installation

```
# apt-get install hubzero-app-submit
# apt-get install hubzero-submit-server
# apt-get install hubzero-submit-distributor
```

Local Configuration

The behavior of submit is controlled through a set of configuration files. There are separate files for defining remote sites, staged tools, multiprocessor managers, legal environment variables, remote job monitors, and ssh tunneling.

Sites

Remote sites are defined in the file sites.dat. Each remote site is defined by a stanza indicating an access mechanism and other account and venue specific information. Defined keywords are

- [name] - site name. Used as command line argument (-v/--venue) and in tool.dat (destinations)
- venues - comma separated list of hostnames. If multiple hostnames are listed one site will chosen at random.
- tunnelDesignator - name of tunnel defined in tunnels.dat.
- siteMonitorDesignator - name of site monitor defined in monitors.dat.
- venueMechanism - possible mechanisms are ssh and local.
- remoteUser - login user at remote site.
- remoteBatchSystem - the possible batch submission systems include CONDOR, PBS, and LSF. SCRIPT may also be specified to specify that a script will be executed directly on the remote host.
- remoteBatchQueue - when remoteBatchSystem is PBS the queue name may be specified.
- remoteBatchPartition - slurm parameter to define partition for remote job
- remoteBatchPartitionSize - slurm parameter to define partition size, currently for BG machines.

- remoteBatchConstraints - slurm parameter to define constraints for remote job
- remoteBinDirectory - define directory where shell scripts related to the site should be kept.
- remoteScratchDirectory - define the top level directory where jobs should be executed. Each job will create a subdirectory under remoteScratchDirectory to isolated jobs from each other.
- remotePpn - set the number of processors (cores) per node. The PPN is applied to PBS and LSF job description files. The user may override the value defined here from the command line.
- remoteManager - site specific multi-processor manager. Refers to definition in managers.dat.
- remoteHostAttribute - define host attributes. Attributes are applied to PBS description files.
- stageFiles - A True/False value indicating whether or not files should be staged to remote site. If the the job submission host and remote host share a file system file staging may not be necessary. Default is True.
- passUseEnvironment - A True/False value indicating whether or not the HUB 'use' environment should passed to the remote site. Default is False. True only makes sense if the remote site is within the HUB domain.
- arbitraryExecutableAllowed - A True/False value indicating whether or not execution of arbitrary scripts or binaries are allowed on the remote site. Default is True. If set to False the executable must be staged or emanate from /apps.
- members - a list of site names. Providing a member list gives a layer of abstraction between the user facing name and a remote destination. If multiple members are listed one will be randomly selected for each job.
- state - possible values are enabled or disabled. If not explicitly set the default value is enabled.
- failoverSite - specify a backup site if site is not available. Site availability is determined by site probes.
- checkProbeResult - A True/False value indicating whether or not probe results should determine site availability. Default is True.
- restrictedToUsers - comma separated list of user names. If the list is empty all users may garner site access. User restrictions are applied before group restrictions.
- restrictedToGroups - comma separated list of group names. If the list is empty all groups may garner site access.
- logUserRemotely - maintain log on remote site mapping HUB id, user to remote batch job id. If not explicitly set the default value is False.

An example stanza is presented for a site that is accessed through ssh.

```
[cluster]
venues = cluster.university.edu
remotePpn = 8
remoteBatchSystem = PBS
remoteBatchQueue = standby
remoteUser = HUBuser
```

```
remoteManager = mpich-intel64
venueMechanism = ssh
remoteScratchDirectory = /scratch/HUBuser
siteMonitorDesignator = cluster
```

Tools

Staged tools are defined in the file tools.dat. Each staged tool is defined by a stanza indicating an where a tool is staged and any access restrictions. The existence of a staged tool at multiple sites can be expressed with multiple stanzas or multiple destinations within a single stanza. If the tool requires multiprocessors a manager can also be indicated. Defined keywords are

- [name] - tool name. Used as command line argument to execute staged tools. Repeats are permitted to indicate staging at multiple sites.
- destinations - comma separated list of destinations.
- executablePath - path to executable at remote site.
- restrictedToUsers - comma separated list of user names. If the list is empty all users may garner tool access. User restrictions are applied before group restrictions.
- restrictedToGroups - comma separated list of group names. If the list is empty all groups may garner tool access.
- remoteManager - tool specific multi-processor manager. Refers to definition in managers.dat. Overrides value set by site definition.
- state - possible values are enabled or disabled. If not explicitly set the default value is enabled.

An example stanza is presented for a staged tool maintained in the HUBuser account on a remote site.

```
[simulator]
destinations = cluster
executablePath = ${HOME}/apps/simulator/bin/simulator.ex
remoteManager = mpi
```

Multi-processor managers

Multiprocessor managers are defined in the file managers.dat. Each manager is defined by a stanza indicating the set of commands used to execute a multiprocessor simulation run. Defined keywords are

- [name] - manager name. Used in sites.dat and tools.dat.
- computationMode - indicate how to use multiple processors for a single job. Recognized

values are mpi, parallel, and matlabmpi. Parallel application request multiprocess have there own mechanism for inter process communication. Matlabmpi is used to enable the an Matlab implementation of MPI.

- preManagerCommands - comma separated list of commands to be executed before the manager command. Typical use of pre manager commands would be to define the environment to include a particular version of MPI amd/or compiler, or setup MPD.
- managerCommand - manager command commonly mpirun. It is possible to include strings that will be sustituted with values defined from the command line.
- postManagerCommands - comma separated list of commands to be executed when the manager command completes. A typical use would be to terminate an MPD setup.
- mpiRankVariable - define environment variable set by manager command to define process rank. Recognized values are: MPIRUN_RANK, GMPI_ID, RMS_RANK, MXMPI_ID, MSTI_RANK, PMI_RANK, and OMPI_MCA_ns_nds_vpid. If no variable is given an attempt is made to determine process rank from command line arguments.
- environment - comma separated list of environment variables in the form e=v.
- moduleInitialize - initialize module script for sh
- modulesUnload - modules to be unloaded clearing way for replacement modules
- modulesLoad - modules to load to define mpi and other libraries

An example stanza is presented for a typical MPI instance.

```
[mpich-intel32]
preManagerCommands = . ${MODULESHOME}/init/sh, module load mpich-
intel32
managerCommand = mpirun -machinefile ${PBS_NODEFILE} -np NPROCESSORS
```

The token NPROCESSORS is replaced by an actual value at runtime.

Environment variables

Legal environment variables are listed in the file environmentwhitelist.dat. The objective is to prevent end users from setting security sensitive environment variables while allowing application specific variables to be passed to the remote site. Environment variables required to define multiprocessor execution should also be included. The permissible environment variables should be entered as a simple list - one entry per line. An example file allowing use of a variable used by openmp is

```
# environment variables listed here can be specified from the command
line with -e/--env option.

OMP_NUM_THREADS
```

Monitors

Remote job monitors are defined in the file monitors.dat. Each remote monitor is defined by a stanza indicating where the monitor is located and to be executed. Defined keywords are

- [name] - monitor name. Used in sites.dat (siteMonitorDesignator)
- venue - hostname upon which to launch monitor daemon. Typically this is a cluster headnode.
- tunnelDesignator - name of tunnel defined in tunnels.dat.
- remoteUser - login user at remote site.
- remoteMonitorCommand - command to launch monitor daemon process.

An example stanza is presented for a remote monitor tool used to report status of PBS jobs.

```
[cluster]
venue = cluster.university.edu
remoteUser = HUBuser
remoteMonitorCommand = ${HOME}/SubmitMonitor/monitorPBS.py
```

Tunnels

In some circumstances access to clusters is restricted such that only a select list of machines is allowed to communicate with the cluster job submission node. The machines that are granted such access are sometimes referred to as gateways. In such circumstances ssh tunneling or port forwarding can be used to submit HUB jobs through the gateway machine. Tunnel definition is specified in the file tunnels.dat. Each tunnel is defined by a stanza indicating gateway host and port information. Defined keywords are

- [name] - tunnel name.
- venue - tunnel target host.
- venuePort - tunnel target port.
- gatewayHost - name of the intermediate host.
- gatewayUser - login user on gatewayHost.
- localPortOffset - local port offset used for forwarding. Actual port is localPortMinimum + localPortOffset

An example stanza is presented for a tunnel between the HUB and a remote venue by way of an accepted gateway host.

```
[cluster]
venue = cluster.university.edu
venuePort = 22
gatewayHost = gateway.university.edu
```

```
gatewayUser = HUBuser
localPortOffset = 1
```

Remote Configuration

For job submission to remote sites via ssh it is necessary to configure a remote job monitor and a set of scripts to perform file transfer and batch job related functions. A set of scripts can be used for each different batch submission system or in some cases they may be combined with appropriate switching based on command line arguments. A separate job monitor is need for each batch submission system. Communication between the HUB and remote resource via ssh requires inclusion of a public key in the authorized_keys file.

Job monitor daemon

A remote job monitor runs a daemon process and reports batch job status to a central job monitor located on the HUB. The daemon process is started by the central job monitor on demand. The daemon terminates after a configurable amount of inactivity time. The daemon code needs to be installed in the location declared in the monitors.dat file. The daemon requires some initial configuration to declare where it will store log and history files. The daemon does not require any special privileges any runs as a standard user. Typical configuration for the daemon looks like this:

```
siteDesignator     = "cluster"
monitorRoot        = "/home/HUBuser/SubmitMonitor"
qstatCommand       = "/usr/pbs/bin/qstat -u HUBuser"
monitorLogLocation = "logs"
```

The directory defined by the combination of monitorRoot and monitorLogLocation needs to be created before the daemon is started. A sample daemon used for PBS batch systems is listed below.

```
#!/usr/bin/env python
#
# Copyright (c) 2004-2010 Purdue University All rights reserved.
#
# Developed by: HUBzero Technology Group, Purdue University
#               http://hubzero.org
#
# HUBzero is free software: you can redistribute it and/or modify it u
nder the terms of the
# GNU Lesser General Public License as published by the Free Software
```

```
Foundation, either
# version 3 of the License, or (at your option) any later version.
#
# HUBzero is distributed in the hope that it will be useful, but WITHO
UT ANY WARRANTY;
# without even the implied warranty of MERCHANTABILITY or FITNESS FOR
A PARTICULAR PURPOSE.
# See the GNU Lesser General Public License for more details.  You sho
uld have received a
# copy of the GNU Lesser General Public License along with HUBzero.
# If not, see .
#
# GNU LESSER GENERAL PUBLIC LICENSE
# Version 3, 29 June 2007
# Copyright (C) 2007 Free Software Foundation, Inc.
#
# ----------------------------------------------------------------------
--
#  monitorPBS.py
#
#  script which monitors the PBS queue and reports changes in job stat
us
#
import sys
import os
import os.path
import select
import time
import popen2
import re
import signal

siteDesignator     = "pbsHost"
monitorRoot        = os.path.join(os.sep,'home','pbsUser','Submit','pb
sHost')
qstatCommand       = "/usr/pbs/bin/qstat -u pbsUser"
monitorLogLocation = "logs"
monitorLogFileName = "monitorPBS.log"
historyFileName    = "monitorPBS.history"

logFile            = sys.stdout
historyFile        = None
activeJobs         = {}
updates            = []
```

```
def cleanup():
   global historyFile

   if historyFile:
      historyFile.close()


def sigGEN_handler(signal, frame):
   global siteDesignator

   cleanup()
   log("%s monitor stopped" % (siteDesignator))
   sys.exit(1)

def sigINT_handler(signal, frame):
   log("Received SIGINT!")
   sigGEN_handler(signal, frame)

def sigHUP_handler(signal, frame):
   log("Received SIGHUP!")
   sigGEN_handler(signal, frame)

def sigQUIT_handler(signal, frame):
   log("Received SIGQUIT!")
   sigGEN_handler(signal, frame)

def sigABRT_handler(signal, frame):
   log("Received SIGABRT!")
   sigGEN_handler(signal, frame)

def sigTERM_handler(signal, frame):
   log("Received SIGTERM!")
   sigGEN_handler(signal, frame)

signal.signal(signal.SIGINT, sigINT_handler)
signal.signal(signal.SIGHUP, sigHUP_handler)
signal.signal(signal.SIGQUIT, sigQUIT_handler)
signal.signal(signal.SIGABRT, sigABRT_handler)
signal.signal(signal.SIGTERM, sigTERM_handler)


def openLog(logName):
   global logFile

   try:
      logFile = open(logName,"a")
```

```
   except:
      logFile = sys.stdout


def log(message):
   global logFile

   if message != "":
      logFile.write("[%s] %s\n" % (time.ctime(),message))
      logFile.flush()


def openHistory(historyName,
                accessMode):
   global historyFile

   if accessMode == "r":
      if os.path.isfile(historyName):
         historyFile = open(historyName,accessMode)
      else:
         historyFile = None
   else:
      historyFile = open(historyName,accessMode)


def recordHistory(id):
   global updates
   global activeJobs

   historyFile.write("%s:%s %s %s\n" % (siteDesignator,str(id),activeJ
obs[id][0],activeJobs[id][1]))
   historyFile.flush()
   updates.append(str(id) + " " + activeJobs[id][0] + " " + activeJobs
[id][1])


def getCommandOutput(command,
                     streamOutput=False):
   child = popen2.Popen3(command,1)
   child.tochild.close() # don't need to talk to child
   childout = child.fromchild
   childoutFd = childout.fileno()
   childerr = child.childerr
   childerrFd = childerr.fileno()

   outEOF = errEOF = 0
```

```
   BUFSIZ = 4096

   outData = []
   errData = []

   while 1:
      toCheck = []
      if not outEOF:
         toCheck.append(childoutFd)
      if not errEOF:
         toCheck.append(childerrFd)
      ready = select.select(toCheck,[],[]) # wait for input
      if childoutFd in ready[0]:
         outChunk = os.read(childoutFd,BUFSIZ)
         if outChunk == '':
            outEOF = 1
         outData.append(outChunk)
         if streamOutput:
            sys.stdout.write(outChunk)
            sys.stdout.flush()

      if childerrFd in ready[0]:
         errChunk = os.read(childerrFd,BUFSIZ)
         if errChunk == '':
            errEOF = 1
         errData.append(errChunk)
         if streamOutput:
            sys.stderr.write(errChunk)
            sys.stderr.flush()

      if outEOF and errEOF:
         break

   err = child.wait()
   if err != 0:
      log("%s failed w/ exit code %d" % (command,err))
      if not streamOutput:
         log("%s" % ("".join(errData)))

   return  err,"".join(outData),"".join(errData)


if __name__ == '__main__':

   if monitorLogFileName != "stdout":
      openLog(os.path.join(monitorRoot,monitorLogLocation,monitorLogFi
```

```
leName))

   log("%s monitor started" % (siteDesignator))

   sleepTime = 10
   pauseTime = 5.
   maximumConsectutiveEmptyQueues = 30*60/sleepTime

   openHistory(os.path.join(monitorRoot,historyFileName),"r")
   if historyFile:
#  lPBS:6760 R
#  --------------------
      records = historyFile.readlines()
      for record in records:
         colon = record.find(":")
         if colon > 0:
            jobState = record[colon+1:].split()
            id = jobState[0]
            status = jobState[1]
            stage  = "Simulation"
            activeJobs[id] = (status,stage)
      historyFile.close()

      completedJobs = []
      for activeJob in activeJobs:
         if activeJobs[activeJob][0] == "D":
            completedJobs.append(activeJob)

      for completedJob in completedJobs:
         del activeJobs[completedJob]

   openHistory(os.path.join(monitorRoot,historyFileName),"a")
   consectutiveEmptyQueues = 0

   toCheck = []
   toCheck.append(sys.stdin.fileno())
   while 1:
      updates       = []
      currentJobs   = {}
      completedJobs = []

      delayTime = 0
      while delayTime  0:
         updateMessage = str(len(updates)) + " " + siteDesignator + ":
" + ":".join(updates)
         sys.stdout.write("%s\n" % (updateMessage))
```

```
         sys.stdout.flush()

      del updates

      if consectutiveEmptyQueues == maximumConsectutiveEmptyQueues:
         cleanup()
         log("%s monitor stopped" % (siteDesignator))
         sys.exit(0)
```

File transfer and batch job scripts

The simple scripts are used to manage file transfer and batch job launching and termination. Examples of scripts suitable for use with PBS are listed here.

File transfer - input files

receiveinput.sh - receive compressed tar file containing input files required for the job. The file .__fileTimeMarker is used to determine what files should be returned to the HUB.

```
#!/bin/sh
#
rm -rf $1
mkdir $1
exitStatus=$?

if [ $exitStatus -eq 0 ] ; then
   cd $1
   exitStatus=$?

   if [ $exitStatus -eq 0 ] ; then
      tar xvzf -
      exitStatus=$?

      touch .__fileTimeMarker
      sleep 1

      date +"%s" > $2
   fi
fi

exit $exitStatus
```

SUBMIT

Batch job script - submission

submitbatchjob.sh - submit batch job using supplied description file. If arguments beyond job working directory and batch description file are supplied an entry is added to the remote site log file. The log file provides a record relating the HUB end user to the remote batch job. The log file should be placed at a location agreed upon by the remote site and HUB.

```
#!/bin/sh
#
cd $1
exitStatus=$?

if [ $exitStatus -eq 0 ] ; then
   case $2 in
      *.pbs)
         JOBID=`qsub $2`
         exitStatus=$?
         if [ $# -gt 2 ] ; then
            logRecord=`date`
            shift 2
            while [ $# -gt 0 ] ; do
               logRecord=${logRecord}'\t'$1
               shift 1
            done
            logRecord=${logRecord}'\t'${JOBID}
            echo -e ${logRecord} >> pbslog
         fi
         ;;
      *)
         echo "Invalid job class $2"
         exitStatus=23
         ;;
   esac
fi

if [ $exitStatus -eq 0 ] ; then
   echo ${JOBID}
else
   echo "-1"
fi
exit $exitStatus
```

File transfer - output files

transmitresults.sh - return compressed tar file containing job output files.

```
#!/bin/sh
#
cd $1
exitStatus=$?

if [ $exitStatus -eq 0 ] ; then
   tar czf - `find . -newer .__fileTimeMarker -not -name . -print`
   exitStatus=$?
fi

exit $exitStatus
```

Batch job script - termination

killbatchjob.sh - terminate the batch job

```
#!/bin/sh
#
case $2 in
   PBS)
      qdel $1
      exitStatus=$?
      ;;
   *)
      echo "Invalid job class $2"
      exitStatus=23
      ;;
esac

exit $exitStatus
```

File transfer - cleanup

cleanupjob.sh - remove job specific directory and any other dangling files

```
#!/bin/sh
#
rm -rf $1
exitStatus=$?

exit $exitStatus
```

Access Control Mechanisms

By default tools and sites are configured so that access is granted to all HUB members. In some cases it is desired to restrict access to either a tool or site to a subset of the HUB membership. The keywords restrictedToUsers and restrictedToGroups provide a mechanism to apply restrictions accordingly. Each keyword should be followed by a list of comma separated values of userids (logins) or groupids (as declared when creating a new HUB group). If user or group restrictions have been declared upon invocation of submit a comparison is made between the restrictions and userid and group memberships. If both user and group restrictions are declared the user restriction will be applied first, followed by the group restriction.

In addition to applying user and group restrictions another mechanism is provided by the boolean keyword arbitraryExecutableAllowed in the sites configuration file. In cases where the executable program is not pre-staged at the remote sites the executable needs to be transferred along with the user supplied inputs to the remote site. Published tools will have their executable program located in the /apps/tools/revision/bin directory. For this reason submitted programs that reside in /apps are assumed to be validated and approved for execution. The same cannot be said for programs in other directories. The common case where such a situation arises is when a tool developer is building and testing within the HUB workspace environment. To grant a tool developer the permission to submit such arbitrary applications the site configuration must allow arbitrary executables and the tool developer must belong the system group submit.