SCons Release Manager

Overview

Just like the CMT Release Manager, the SCons Release Manager is split into several components. These components are the batch submission system, the workflow system, and the release manager system. Unlike the CMT Release Manager, the SCons Release Manager is not written in perl. It is written in C++ relying mostly on the Qt library. Additionally, it depends on the Boost Spirit library, and the stx-exparser library. The Qt library is used for most platform independent interactions. The Boost spirit and stx-exparser are used on the linux side only for linux components only (in particular, for a Workflow application only).

Qt

A major component of the SCons infrastructure is the Qt library. For linux and the mac the source code is obtained from the trolltech website and compiled. In both cases it's compiled as a static library. For windows the installation is a bit more complicated. The windows installation doesn't come with the mysql plugins pre-compiled. As a result it needs to be compiled after installing Qt for windows. From the trolltech webpage one needs to download the LGPL /Free version for windows. There are several choices to download. The appropriate one is the Framework Only version for Windows. This version is complied with mingw so the machine it's installed on will need mingw. The installer for Qt will take care of installing it but it should **NOT** be installed on any of the machines that are used by the ReleaseManager as SCons might get confused with mingw and visual c++ both installed.

Once the Qt installer is done, the mysql plugin needs to be compiled for Qt. The instructions for compiling Qt 4 MySQL drivers with MinGW are provided here http://qt-project.org/doc/qt-4.8/sql-driver.html (Note: I haven't actually tested these instructions but the old link that was here points to an unregistered domain and this set of instructions is from the Qt sites directly.)

Batch Submission System

Just like its CMT counterpart, the Batch Submission System is used to submit jobs to lsf, monitor the jobs, and notify the layers above of finished jobs. The information for the Batch Submission system is stored in a MySQL database on mysql-node03.slac.stanford.edu. Additionally, Batch Submission System consists of a c++ library and two applications which obtain their information from the database.

Just like the CMT counterpart, the Batch Submission System accepts jobs to execute from the upper layers of the system, executes them on lsf, and notifies the upper layer of state changes.

Database

The MySQL database for the Batch Submission System can be accessed by the **rd_lsf_u** user and the **glastreader** user. The database name is **rd_lsf**. The database contains the following tables

- **arguments** – This table contains the arguments to be passed to the program when executing on lsf.
- **callback** – This table contains the details on which state changes the upper layer should be notified.
- **callbackArgs** – This table contains arguments to pass to the callback program when executing it.
- **command** – This table contains the command to execute on lsf.
- **job** – This table contains contains meta data, such as when it was registered, on commands to be executed on lsf.
- **lsfOptions** – This table contains any options to be passed to lsf when submitting.
- **output** – This table contains the output of the lsf job.
- **run** – This table contains the details of a job's attempt at executing on lsf.
- **settings** – This table contains global settings for the Batch Submission System.

Applications

The Batch Submission System consists of two applications. The lsf Daemon and the lsf callback application. The daemon is setup to run on fermilnx-v03 (a virtual machine) via trscron (trscron is used instead of regular cron because the LSF processes need AFS tokens and normal cron doesn't provide them). The daemon runs for an hour and is then retriggered.

It checks the database for new jobs. Jobs are submitted to LSF in suspend mode. For each job, there are three LSF tasks submitted. The first task is the actual command to be executed. The second task is a call to the lsf callback application with the condition that it be executed as soon as the first task starts execution. The third task is a call to the lsf callback application with the condition that it be executed as soon as the first task finishes execution. All LSF tasks are submitted and initially put into suspend mode. If all three tasks are submitted successfully, the jobs are changed from suspend to pending mode so they actually start execution. If at any time anything goes wrong, the LSF tasks are killed and the run in the table is marked as failed. The job is put back into the queue to be attempted at a later time as a new run.

The lsf callback application is called once when the LSF task starts and once when it finishes. When started, it updates the **run** table with the information about the start time. Additionally, it checkes the **callback** table to determine if the higher layers wish to be notified when the job has started. If so, it calls the callback command to notify the higher levels. When called for a task that has finished. The callback system checks the LSF output, parses some of the information, like the return code, and stores that information in the **run**, **job**, and **output** tables. Additionally, it checks for any callback information for the higher layers.

Settings

The batch submission settings are controlled by a settings table with a bunch of name/value pairs. The valid names that can be specified is currently not documented anywhere except in code. The code can be viewed in CVS

http://www-glast.stanford.edu/cgi-bin/viewvc/grits-cpp/src/lsf/lsfDaemon/JobSubmitter.cxx?revision=1.2&view=markup

The function JobSubmitter::callBsub() is the one that parses these name value pairs and creates the correct bsub command line options.

Workflow

The Workflow system is a rule based script execution system. Each script is considered a stage in the workflow. The workflow moves from one stage to the next by evaluating rules set forth for each stage. The rules and stages for the workflow are stored in a database on mysql-node03. The workflow consists of a static library and a single executable. The library is used by other systems that wish to use the workflow for initial submission and controlling various settings of the workflow runs. The executable is run after each stage of the workflow finishes execution in the batch submission system described above. It computes which new stages need to be executed based on the rules described for the finished stage.

Workflow Database

The database is stored on **mysql-node03** and the database name is **rd_workflow**. The database contains the following tables

```
+-----------------------+
| Tables_in_rd_workflow |
+-----------------------+
| batchOpts             |
| batchOptsOverride     |
| conditions            |
| run                   |
| runArgs               |
| runScripts            |
| runScriptsOverride    |
| runSettings           |
| settings              |
| workflowScripts       |
| workflows             |
+-----------------------+
```

Each table performs a unique function as follows

- **settings** – This table contains name/value pairs of settings for the workflow system.
- **workflows** – This table contains a list of workflows registered.
- **workflowScripts** – This table contains a list of all the scripts (aka stages) for a particular registered workflow.
- **conditions** – This table lists all the conditions for a particular stage and what next stage to execute if condition evaluates to true.
- **batchOpts** – This table contains a name/value pair of batch options to pas to the batch submission system for a particular stage.
- **run** – This table contains the information for an actual execution of a workflow.
- **runScripts** – This table contains which stage(s) are currently executing for a particular run.
- **runScriptsOverride** – This table contains a few override settings of default stage settings such as which batch queue to execute.
- **runSettings** – This table contains name/value pairs of settings to override for a particular run.
- **runArgs** – This table contains the arguments to pass to the stages for a particular run.
- **batchOptsOverride** – This table contains name/value pairs of batch options to override for a particular stage in a particular run.

Library

The static library for the workflow system contains a set of functions that allows C++ programs to create new workflow runs and for specifying settings for these runs. The source code and the available functions can be viewed in CVS and is also compiled and available at ~glastrm/grits-cpp/src. The library makes use of the Qt libraries and uses qmake as its build system. The Qt library currently used is version 4 and it's installed in ~glastrm/externals which is a set of symlinks using AFS @sys variable to the appropriate platform installation in nfs.

Executable

The workflow system has a single executable called **workflowCallback**. It is executed by the batch submission system when each stage starts executing and stops executing. When called for the start of the execution of a stage, it simply updates the mysql tables to indicate the start time. When executed to indicate the stop of execution of a stage, additionally it also checks the condition table to determine which, if any, other stages need to be executed. If new stages for execution are found they are submitted to the batch submission system and the cycle is repeated until a particular run has no more stages to execute.

Release Manager

Just like in the CMT version, the Release Manager consists of a few scripts that are registered in the workflow system as various stages. Conditions are setup in the workflow stages for which stages to execute next. Additionally, the Release Manager consists of several mysql tables.

Release Manager Database

The Release Manager database is stored on **mysql-node03**. The database name is **rd_releasemgr**. The database contains the following tables:

```
+---------------------------+
| Tables_in_rd_releasemgr   |
+---------------------------+
| build                     |
| buildPackage              |
| deletedBuild              |
| entry                     |
| extLib                    |
| os                        |
| outputMessage             |
| package                   |
| settings                  |
| subPackage                |
| subPackageCompileFailures |
| variant                   |
| versionType               |
+---------------------------+
```

The explanation of each of the tables is as follows

- **build** – This contains the platform dependent information for a particular build.
- **buildPackage** – This contains the platform independent information for a particular build.
- **deletedBuild** – This table contains has same structure as the **build** table but contains the entries for builds that have been completely removed by the *eraseBuild* program.
- **entry** - this table was migraged from glastdb and hold the log entries from the operation of all the RM processes. Logs are kept for 5 days and then removed to prevent the database from filling up.
- **extLib** – This contains the external libraries used for a particular build.
- **os** – This contains a list of operating systems currently supported.
- **outputMessage** – This contains the checkout or compile output for a particular build.
- **package** – This contains a list of checkout packages currently supported.
- **settings** – This contains both global and build specific settings for the Release Manager.
- **subPackage** – This contains a list of subPackages and their compile status for a particular build.
- **subPackageCompileFailures** – This contains the location of specific compile failures in the compile output for a particular subPackage.
- **variant** – This contains a list of supported variants.
- **versionType** – This contains a list of supported versionTypes.

The **os**, **variant**, **versionType**, and **package** tables define a list of possible combinations of what the Release Manager can compile or keep track of. When new packages, oses, etc. are created they need to be inserted in these tables. Each of these tables will create automatic unique IDs that are referenced by the other tables. The settings table references both platform independent settings and platform dependent settings. The platform independent settings have NULL values for the columns that reference the os or variant tables and only include values for the package and versionType tables. The platform dependent settings have values that refer to os, variant, versionType, and package tables.

Executables

The executables and a short description of what they perform is as follows

- **checkoutBuild** – This executable checks out code from cvs in preparation to building it.
- **cleanBuild** – This executable erases parts of a build that are not needed for execution (source code and temporary files).
- **compileBuild** – This executable builds the code checked out with checkoutBuild.
- **createDoxygen** – This executable generates doxygen output for a build.
- **deleteBuild** – This executable is an interactive program to trigger the RM to erase a build. By default, the entry will remain in the database.
  ex. deleteBuild --package ScienceTools --versionType Integration --version 3366 --os Windows-i386-32bit-vc90 --variant Debug --complete
  versionType may be Integration "Release Candidate", or Release
  The --complete option erases the build and also removes the entry from the database. This is used when one wishes to re-trigger a build.
- **eraseBuild** – This executable erases everything belonging to a build and marks the build as hidden in the database. This is called by deleteBuild and most likely never need be called directly by a user. As this is the program called by deleteBuild to actually do the remove, it also supports the --complete option.
- **finishBuild** – This executable marks the build as having finished.
- **releaseManagerDaemon** – This executable runs in the background submitting new builds to the workflow.
- **testBuild** – This executable runs the unit tests for a particular build.
- **triggerBuild** – This executable can be used to trigger a new build.
- **createReleaseBuild** – This executable creates the tar.gz/zip files of the release and externals for the installer.

The releaseManagerDaemon runs for 6 hours on fermilnx-v03 before shutting down. A cron job on fermilnx-v03 restarts the process every 6 hours. Regular cron is used as no AFS tokens are needed for this daemon and the use of trscron (changing the trscrontab entries) requires the glastrm account to have a password which the system administrators would rather not have. The process used to run for 24 hours but changes to the MySQL database backend resulted in timeouts after 8 hours so the run period was shortened to accommodate this issue.

While running, it checks for new builds to submit to the workflow based on the rules specified in the settings tables. It is capable of submitting builds for all the combinations that are allowed by the os, variant, versionType, and package tables as long as appropriate settings exist in the settings table.

All executables take an argument of --buildId <buildId>. The buildId references the unique ID stored in the build table. These executables are in /u/gl /glastrm/grits-cpp/bin on slac linux.

Notification

The Release Manager notifies users of failures when executing the finishBuild executable. Similar to the CMT RM, the finishBuild executable checks the database for any registered failures and looks up the author's information in the package SConscript file. Unlike the CMT RM, the finishBuild executable attempts to bundle as many failures together for a particular author as it can. The goal is to only send a single email per build to a person for all the packages that belong to the author which contain errors.

Archiver

Archiver episode - 20140402

The archiver is another system that works on top of the Workflow system. Just like the Release Manager, it has a trigger script registered with the workflow system that triggers when a new archive job is started. Similarly, the archiver contains database entries and a web page for viewing the information.

Database

The database for the Archiver is stored on glastDB.slac.stanford.edu with the database name of **Archive**. It contains the following tables:

- **archiveContent** - The files archived.
- **pending** - Contains pending archive entries.
- **tarFiles** - Contains a list of tar files that belong to a single archive job and their locations in mstore/gstore/astore.
- **task** - The main table containing the archive job.

All the tables are tied together via an id field defined by the **task** table. Tar files are split at a size defined by the user at archive creation time and the list of tar files belonging to a single task is stored in the **tarFiles** table.

Scripts

The scripts for the Archiver are stored in **/u/gl/glast/infraBin/Archiver**. The scripts located there are:

- **archive.pl** - The script invoked by the workflow when archiving.
- **archiver.pl** - This script is the controller script with which users can archive, restore, delete jobs.
- **delete.pl** - This script is invoked by the workflow when deleting archived files.
- **determineTask.pl** - This script is invoked to determine if a job is for archiving, deleting, restoring, etc.
- **finish.pl** - This script is invoked as the last script by the workflow to cleanup the database and mstore/gstore/astore.
- **restore.pl** - This script is invoked for restore operations.
- **trigger.pl** - This script is invoked by the workflow to determine if new archiving tasks are pending.
- **verify.pl** - This script is invoked for verify operations.

All of these scripts are mostly frontends to the mstore/gstore/astore applications. The use the expect perl module to programmatically control tar which expects interactive input for creating and splitting tar files.

Here is an example how to archive a file CLHEP-1.9.2.2.tar.gz in directory /nfs/farm/g/glast/u05/extlib/tiger_gcc33:

/afs/slac.stanford.edu/u/gl/glast/infraBin/Archiver/archiver.pl --module "Manual" --callback "user:kiml" --path "/nfs/farm/g/glast/u05/extlib/tiger_gcc33" --user glast --name "u05.extlib.tiger_gcc33.CLHEP-1.9.2.2.tar.gz" --file CLHEP-1.9.2.2.tar.gz --method mstore archive

And to restore it to /nfs/farm/g/glast/u30/tmp:

/afs/slac.stanford.edu/u/gl/glast/infraBin/Archiver/archiver.pl --module "Manual" --callback "user:kiml" --path "/nfs/farm/g/glast/u30/tmp" --user glast --name "u05.extlib.tiger_gcc33.CLHEP-1.9.2.2.tar.gz" --file CLHEP-1.9.2.2.tar.gz --method mstore restore

Notification

The RM scripts notify the users of problems encountered during checkout, compile, or unit tests. This notification is done by the **finish.pl** script. It checks the database for every package that belonged to a build to determine if the package had any errors during checkout, build, or unit test compiles. If there were errors, the script checks who the author is for the failure and notifies the author. Additionally, all the failures are accumulated and sent to a general failure mailing list as specified by the settings page.

Webpages

The web page for the Archiver is https://www.slac.stanford.edu/www-glast-dev/cgi/DiskArchive. It is controlled by the SLAC web server. The information is displayed by a bunch of perl scripts located in /afs/slac/g/www/cgi-wrap-bin/glast/ground/cg/archive.

Old Style Tagger

With the shut down of automated CMT builds, the old style tagger has also been shut off (June 2013) and its use is now deprecated. However, the material is kept here for reference.

The CVS repository is currently set to deny any tags that do not conform to the format packageName-XX-YY-ZZ when the following rules apply:

1. The tagging user is not glastrm
2. The package being tagged is not in the users directory

These rules are enforced by the tagger script that's located in /nfs/slac/g/glast/ground/cvs/CVSROOT/tagger.

The CMT Release Manager still expects old style tags of the form vXrYpZ so the tags above need to be converted to the old style tags. This conversion is done in two steps. First, the tagger script writes a history of tags to a file located in /nfs/slac/g/glast/ground/cvs/CVSROOT/tagHistory. The second part of this is a cron job running every minute on glastlnx14. It executes a perl script located in ~glastrm/oldStyleTagger.pl. This script reads the tagHistory file and determines what tags have occurred since the last time it ran (1 minute ago). It determines which packages where tagged with the new style tags and converts them to old style tags. It then applies these old style tags to the package. This cron job has to run as the glastrm account otherwise the two rules above that deny old style tags will apply and the tag will be denied. There's also a testOldStyleTagger.pl which is identical to oldStyleTagger.pl except for not actually executing cvs tag commands. It is used for testing out what the oldStyleTagger.pl would do for debugging purposes.

How to turn off the CVS tagging (in the case of an outage)

- cvs co CVSROOT/tagger
- edit CVSROOT/tagger
  insert a useful print statement and exit(1)
- cvs ci CVSROOT/tagger

It is also a good idea to temporarily point the stag binaries at a "do nothing" script to prevent further tagging attempts via stag. See:
/afs/slac/g/glast/ground/applications/install
i386_linux26/usr/i386_rhel50/ b in and
amd64_linux26/usr/i386_rhel50/ bin

To re-enable tagging, just undo the changes made above.

Miscellaneous Details

This section provides links to various sub-pages describing random bits of SCons lore.

- Database Keys and IDs and How to Connect Them
- Current Build Infrastructure
- Jenkins use in the Release Manager
- Notes on building the RM tools on supported platforms
- Adding a new package or OS to the RM
- Common maintenance and monitoring tasks