SCons Release Manager Functions and Structure

RM, the Narrow View

This section concerns only the code under grits-cpp/releaseManager and the databases it talks to.

Application-level functions used by Fermi

Automated functions

- Automatically make ("HEAD") builds for configured OSes when a new release candidate tag is created
- Automatically make ("Release") builds for configured OSes when a new release tag is created
- Automatically make ("LATEST") builds for configured OSes when a new subpackage tag is created

where "making a build" means

1. fresh checkout of proper version of code from repository
2. compilation, creating libraries and executables (SCons)
    a. also copy externals used to Installer location if not already there
3. creation of tarballs for distribution (SCons)
4. running package test programs
5. creating Doxygen output
6. notifying package owners of failures

Among the RM code are executables with names like "checkoutBuild", "createDoxygen", etc. Each of the steps above involves calling such a program. For those steps labeled (SCons) the bulk of the work is done by invoking SCons.

Results of various kinds are saved for later retrieval and display.

Manual functions

The functions above invoke various executables to accomplish their tasks. These executables may be invoked by hand if, for example, some part of the build output needs to be generated or regenerated. The RM code also includes some additional executables which may be invoked manually. Typical tasks which one might want to do manually include

- Make a build for a particular os-container-variant combination which is defined in the db but not automatically triggered
- Delete an old build, typically with intention of retriggering

Archiver

A separate application, the Archiver, makes use of the lower levels (lsf, workflow engine). Under normal circumstances it is invoked periodically via trscron and runs an ancient Perl script. Additional scripts may be run by hand to, e.g. archive or restore a particular file.

RM Structure

RM (excluding Archiver) accesses three MySQL databases: rd_lsf, rd_workflow and rd_releasemgr.

- rd_lsf is used for interactions with the lsf batch system.
- rd_workflow is used to properly sequence jobs needed for a particular operation. For Jenkins workflows there is just one entry describing the launch of a Jenkins job; the details (first checkout code, then build, etc.) are handled elsewhere. For OSes not directed to Jenkins, there are entries in the workflow db describing what has to happen and what steps any given step depends upon.
- rd_releasemgr is by far the most complex of the three. It contains all the information needed to configure builds by os, container (ST, GR, etc.), build type (Release, Release candidate, LATEST) and variant (Debug, Opt) and also information about the status of particular builds (by os, container, build type and tag version).

For each of these databases there is a part of the RM code which talks to it. The lsf and workflow parts have been untouched for 5 years and even those updates were very minor. They haven't seen any significant activity for 7 or 8 years. The lsf and workflow parts of the code only need to run where RM itself is running, currently rhel6. The rest must run on any platform where builds are supported (currently rhel6 and rhel7).

There is considerably more, and more complicated, code talking directly to rd_releaseMgr. There have been maintenance updates (not exactly frequent, but steady at a low level) right along. Speaking very roughly, the function of this code is, based on data in rd_releaseMgr and on information (about package structure and tags) gleaned from the code repository, to decide what builds are necessary to create (or delete). It then does the necessary operations, including updating the database and, in the case of LATEST builds, creating tags in the code repository.

If we used Jenkins for all builds we could perhaps vastly simplify the workflow section of the code. If we also stopped using lsf maybe we could replace both the workflow and lsf code with something much more streamlined. For the rest, I don't see any clear path to something significantly simpler.

Active Builds

Packages being built currently by RM are summarized in this Current Build Infrastructure page. Future plans include a new container package, ScienceTools_User (ROOTless subset of ScienceTools) and possibly support for a modern Mac OS for ScienceTools and ScienceTools_User.

The RM Ecosystem

In addition to the core functions described above, there are several others which depend to some degree on RM being constituted more or less as it currently is. For the most part this boils down to a dependence on one or both of the rd_releaseMgr database and tagging conventions.

Release Manager results on the Web

Information displayed comes from RM database. It includes status of all builds, compile output, test output, red-dot display (comparison of most recent release build, release candidate build, and LATEST build).

Installers

Information needed to satisfy clients comes from RM database. Build products transported to client are created by RM.

Tagging

The only tags RM creates are for LATEST builds (use most recent tag for each subpackage), but RM expects both package tags and container-wide tags to be of a certain form. We currently have a tool (C++ program) to help users create subpackage tags of the proper form and another (Python script) one for creating container-wide tags for releases and release candidates. We'll need something for the equivalent tasks which talk to git rather than CVS. (Assuming we don't change the required tag form these tools have nothing directly to do with the fate of RM; the need for new tools is driven by the move to git.)