3rd party libraries

Apps-2017 build notes

Notes on MacOS:

src-2017:

All source code and build scripts are kept on glade: /glade/p/DASG/VAPOR/third-party/src-2017 . Each package has its own folder, for example, /glade/p /DASG/VAPOR/third-party/src-2017/qt-4.8.7 . Each folder contains the source code for that library, as well as a build script.

Versions:

Third party library versions are in the text file on glade: /glade/p/DASG/VAPOR/third-party/src-2017/versions.

Note on python packages: Python packages are installed with pip. More specifically, these packages include numpy, scipy, matplotlib, seedme, and their dependencies.

Note on QT: QT version is 4.8.7, which could be found at: https://download.qt.io/official_releases/qt/4.8/4.8.7/ . To successfully compile this version of QT on MacOS 10.11, the following patch needs to be applied: https://gist.github.com/ejtttje/7163a9ced64f12ae9444 .

Note on FTGL: On mac system, install GNU grep and sed to compile successfully. With homebrew, these two tools are **homebrew/dupes/grep** and **gnu-sed**.

Configure and build procedures:

Most packages need to go through 4 steps to installed: 1) extract the tarball; 2) configure; 3) make; and 4) make install. Take qt as example, the first step would result in a folder "qt-everywhere-opensource-src-4.8.7". The second step would be cd'ing into qt-everywhere-opensource-src-4.8.7, and type "make -f ../Makefile.NCAR configure." The third step would be "make," and the 4th step is "make install."

Notes on Linux:

Sam's linux box seems to need two more dependencies: doxygen and libexpat. If VAPOR still complains that it's not able to find libexpat.so.0 after installing libexpat from the repo, locate libexpat.so.1 (/lib/x86_64-linux-gnu/libexpat.so.1 on Ubuntu) and create a symbolic link to it with the name libexpat.so.0.

Old Notes

Shared libraries present numerous problems under Linux, Mac OS, and most likely other UNIX operating systems. One of the most problematic issues is that an application may be linked against one shared library at link time. However, at run time the run time loader may find a different version of the same shared library in some other search path.

Three important cases need to be addressed with VAPOR to ensure the correct shared libraries are always loaded at run time.

Case 1: running within source tree (make all): It should be possible to run vapor executables from within the source tree without setting environment variables (i.e. running vapor-setup).

- *correct* 3rd party libraries should be loaded at run time (*correct* => the libraries the executables were built with)
- VAPOR's internal libraries (e.g. libvdf) should be found

Addressing the two points above can probably be solved by using the ld -rpath option.

Case 2: Running from installed source (make install): Once someone installs vapor by running 'make install' the correct libraries should be found **after** sourcing vapor-setup.csh, which simply sets the LD_LIBRARY_PATH (DYLD_LIBRARY_PATH on Mac) to point to the lib directory at the root of the install path.

- The tricky issue here is that VAPOR's internal libraries must be installed and the executables should than load the internal libraries from the installed path, not the source tree from which vapor was built. chrpath can be used under linux to delete hard-coded shared library paths. intall_name_tool may offer similar capability on the Mac.

Case 3: Running from a binary installer: The binary installer (vapor-install.csh) copies the vapor executables, internal libraries, **and** 3rd parth libraries to a new location. It also modifies the vapor-setup.csh to point to this new location (i.e. correctly set the LD_LIBRARY_PATH). Hence, it must be possible to change the installed directory of vapor executables and have all libraries correctly found.

Helpful tools

ldd: Under linux ldd can be used to show which shared libraries an executable or a shared library will load at run time under the current environment

otool: On Mac OS this tool (when invoked with the -L) option is similar to ldd. However, the library paths listed are default paths used if not overridden by environment variable settings (e.g. DYLD_LIBRARY_PATH). The only way to know for certain on the Mac which shared library will be loaded at run time is to set the DYLD_PRINT_LIBRARIES environment variable and then run the executable.

install_name_tool: Under Mac OS can be used to change the path of the shared library that an executable (or library) depends on

chrpath: Under Linux can be used to change or delete RPATH and RUN_PATH settings in an executable. Note this command is currently installed only on storm0. Note that if your new RPATH can not be longer then the existing RPATH string.

Proposed solution

Case 1:The goal in this case is to allow developers (us) to run the executables without having to set environment variables, etc., and have the correct shared libraries loaded - the loader should pick up VAPOR (internal) libraries from inside the build tree, and 3rd party libraries from wherever they live on the build system.

Thus the proposed solution is to simply embed the absolute search path to both internal and 3rd party libraries into the executable. Under Linux and Mac OS this can be accomplished with the ld -rpath and -dylib_install_name command line options, respectively.

Note: the use of rpath is not an ideal solution in that it doesn't prevent the possibility of a name collision between two libraries located in different directories specified by the RPATH. However, there does not seem to be a portable solution (across all platforms) to this problem (on the Mac it looks like the default behavior of the linker is to include the full path to the library in the executable).

Note: rpath is not supported under Mac OS 10.4. However, later versions (10.5 and 10.6) appear to support this option, it's implementation and behavior is somewhat different than under Linux. We might consider using rpath for both Linux and Mac OS after support for Mac OS 10.4 is dropped.

Case 2: This is the 'make install' case and is primarily targeted at customers wishing to build vapor from source. This case is complicated by the fact that we can't assume customers have 'chrpath' installed on their build platforms (not sure about install_name_tool for Mac users). Thus the proposed solution is to simply relink the libraries with no path information set. Hence under Linux the ld -rpath option would not be used during linking. For Mac OS the target of the -dylib_install_name option would be set to the library name without a preceding path (or perhaps set to @executable_path/<lib_name>).

Note: With no rpath/install_name the LD_LIBRARY_NAME (DYLD_LIBRARY_NAME) on Mac) environment variable must be set for the run time linker to find dependent shared libraries -- users will be forced to source vapor-setup.csh. This is not a problem as the current build system requires this as well.

Case 3: This case corresponds to the current VAPOR "make install-dep" target. The challenge here is that we are preparing binaries that could be installed (copied) anywhere on the customers machine. The solution here is no different than for case two -- linking without including path information in the resulting executable (or library on the Mac). However, in order to support Mac "applications" (.app files) the install_name embedded in all VAPOR executables and distributed libraries (both VAPOR internal and 3rd party) must be set to @executable_path/<libname>. This is no different than what is presently done, and is accomplished as a post-processing step with 'install_name_tool'.

Note: As case 3 is only required/supported for VAPOR developers there is no problem relying on install_name_tool, which may or may not be installed on a customer's platform.

Shared Library Findings:

Mac OS:

There have been several changes in how dynamic libraries are accessed and created between the various versions of Mac OSX. The following is an explanation of these differences and how the operating system uses dynamic libraries. This discussion is going to be limited Mac OSX 10.4 (Tiger) and later.

First concept to cover is the difference between the name and install_name of a dynamic library. The name is the filename, without any path information, of the the library, i.e. libfish.dylib. Install_name is a value (LC_ID_DYLIB) which is embedded within the library and can be displayed using otool -l. This value can be just the name, the filename with a fully qualified path, the filename with a relative path, or the filename with one of the macros (@executable_path, @loader_path, or @rpath) . The install_name is set using using the linker (ld) option “-install_name name” for MacOS 10.5 (Leopard) and MacOS10.6 (Snow Leopard) system and the option “-dylib_install_name name” for MacOS 10.4 systems when creating the dynamic library. If there is no value for the install_name passed to the linker, the linker will use the output filename as the install_name. In both cases you will need the -dylib option for the linker in order for the output file to be a dynamic library. The value of the install_name can be changed after the library has been created using the command install_name _tool with either the -id or -change options, this command is available on all Mac OS X versions. An example command of the linker to create a library:

ld fish.o -dylib -install_name @rpath/libfish.dylib -o libfish.dylib -lc

[kendall, Can the new install_name be of arbitrary length?]

It is best that the install_name end with the library's name.

During the creation of the executable which uses your dynamic libraries, the linker makes use of both the name and install_name of the library. During linking phase the linker will use the values from any -L and -l options to determine the name and location of dynamic library that is being used. If all the symbols being requested are resolved, the linker will embed the install_name from that library into the final executable file. Using the otool -L command on the executable file, the install_name for every dynamic library that is required to the program to run will be displayed. Note: otool -L doesn't display recursive dependencies: if the shared libraries themselves are dependent on other shared libs, running otool -L on the executable will not show these recursive dependencies.

When a program is running there are two paths that the runtime loader uses to find the dynamic library it needs. The choice of which path is used depends on if the install_name for that library to be loaded that is embedded in the executable file is just the name of the library or has any path information, this will include any macros. Depending on which of the two cases the runtime loader has a very specific order which it uses to search for a library. The order of precedent for searching in the first case, the install_name is the filename of the library, no path information:

1. DYLD_LIBRARY_PATH This is an environment variable that contents a colon delimited list of directories.
2. The process's working directory
3. DYLD_FALLBACK_LIBRARY_PATH This acts like an environment variable but does not show up when display your environment variables. The default value is : $HOME/lib:/usr/local/lib:/usr/lib .

The documentation did state that the environment variable LD_LIBRARY_PATH had a higher precedent then DYLD_LIBRARY_PATH in this search case, but my testing showed that it was not being used for searching in this case.

The order of precedent for searching in the second case, the install_name has path information:

1. DYLD_LIBRARY_PATH
2. The given path This can include either a relative or absolute path to the library. If there is a macro being used, the value is substituted in and the new path is searched. Examples are ../lib/libfish.dylib, /opt/zoo/lib/libbird.dylib, or @loader_path/../lib/libbird.dylib .More on macros in the next section.
3. LD_LIBRARY_PATH This is an environment variable that contents a colon delimited list of directories.
4. DYLD_FALLBACK_LIBRARY_PATH

The documentation did state the the value LD_LIBRARY_PATH was not used in this search, but my testing showed the above order.

There are three macros values that are used in determining the location of a dynamic library at run time:

- @executable_path

The value of the path to the program that is loading the library is substituted.

- @loader_path

The value of the path of the object that is loading the library is substituted. This could be the path of a dynamic library loading another library. This was introduced in MacOS 10.4.

- @rpath

This was introduced in version MacOS 10.5 and works more like rpath in Linux. Each rpath value that is embedded in the executable is substituted and then searched. The -rpath option to the linker is used to add one path at a time to the list of rpaths to be searched. For example, passing -rpath /usr/farm /lab -rpath /usr/zoo/lib will add both directories, or passing -Wl,-rapth,/usr/zoo/lib on the compiler option will also add this directory to the executable's rpath. Using otool -l will show each rpath (LC_RPATH) value of an object file. In MacOS 10.5 there is no way to change or add the rpath values of a files once it has been created, in MacOS 10.6 the install_name_tool has added several new options to work with the rpath values.

A few of the problem that may occur are: There exist another library with the same name that is found earlier in the search order then the library that you wanted the program to use. The install_name of a library has the path and name of either another or nonexistent library.

References:

http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/DynamicLibraries/100-Articles/DynamicLibraryUsageGuidelines.html#//apple_ref/doc/uid/TP40001928