EMERGE: ESnet/MREN Regional Grid Experimental NGI Testbed
NCSA subaward
DOE DEFC02-99ER25406 (1-5-28106)

Eric Blau, Michael Bletzinger, Randy Butler

1. Abstract

As part of the EMERGE Gird Testbed activities NCSA proposed to develop a standard grid service package, and lead the effort to deploy grid technologies to the Emerge testbed sites. Funding for this project was limited to only the first year of what was to be a three-year project. As a result NCSA concentrated on the development of the Grid Services Package.

NCSA successfully developed the Grid Packaging Toolkit (GPT), which was designed to simplify deployment and installation of Grid software at the five Emerge testbed sites. GPT fulfills the grid deployment requirements for such things as heterogeneous platform support, service perquisites, installation procedures, documentation and training, site-specific integration concerns, and providing for standard deployment of grid software. Within the scope of the Emerge project NCSA was able to develop the core GPT capabilities and to begin to apply GPT towards the packaging of the Globus Toolkit, meeting the goal of an easy to deploy Grid Service Package.

Since the conclusion of the Emerge grant, NCSA has been able to successfully apply GPT to the Globus Toolkit and indeed the most current version of Globus, version 2.0, is packaged via GPT. Previous Globus Toolkit versions were a monolithic distribution of many interrelated components. The use of GPT has made it possible for organizations to construct both source and binary distributions of the Globus Toolkit components in which they are most interested. GPT makes it possible to release upgrades for individual Globus Toolkit components without having to release the entire Toolkit.

NCSA has continued to advance GPT through NSF and NASA IPG funding, and extended its capabilities further. GPT is the chosen software packaging solution for NSF's recently announced NSF Middleware Initiative (NMI).

2. Introduction

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

There are many organizations trying to deploy Grid software in a uniform way across heterogeneous landscape of different operating systems, site configurations, and administrative boundaries. These developing Grids are largely configured with the Globus Toolkit software. The goal of the packaging effort was to construct a framework and a set of Globus Toolkit packages, which could be used to create tailored distributions that fit the specific organizational needs. Another goal of the packaging effort was to apply this new packaging technology to the Globus Toolkit and thereby allowing individual pieces of what was previously a monolithic distribution to be released as separate components.

Previously the Globus Toolkit consisted of many components was inflexible in that it did not support the deployment of subsets of Globus. To alleviate this problem the Globus components were divided into packages. Using the new packaging framework, organizations are able to construct source and binary, organization specific distributions of the selected packages they are interested in. Under the new packaging framework, it is possible to release updates to an individual component without releasing updates to all other components. Additionally, an individual component can be built and tested without configuring and building unrelated components. Thus the packaging framework substantially increases the efficiency of the development and release process.

GPT is a collection of packaging tools built around an XML based packaging data format. This format provides a straightforward way to define complex dependency and compatibility relationships between packages. The tools provide a means for developers to easily define the packaging data and include it as part of their source code distribution. Binary packages can be automatically generated from this data. The packages defined by GPT are compatible with other packages and can easily be converted.

- Developer Tools
  GPT provides a build system in conjunction with the globus_core package from the Globus Toolkit. GPT has tools to convert a source distribution into a GPT based package. It also provides patch-n-build capability similar to RPM spec files for those sources distributions that need to retain their own build system.
- User Tools
  GPT provides tools that enable collections of packages to be built and/or installed. It also provides a package manager for those systems that don't have one.

The GPT software is currently in beta release and available for download from ftp://ftp.ncsa.uiuc.edu/aces/gpt/.

The rest of this report lays out the design of a packaging framework that can be used to enable the packaging of complex Grid software and use the requirements of the Globus Toolkit to highlight an example use of GPT. We will introduce packaging concepts, and discuss our chosen strategies and policies for dividing large pieces of software into logical packages. We will describe the different types of binary packages, as well as the distinctions between source packages and binary packages. We will also describe the mechanism by which the packaging system manages complex build environments, and what it means to have a Globus installation in the context of the packaging system.

Finally, we will describe the metadata needed by the packaging system, for each package type, to allow the system to effectively manage the packages.

3. Grid Packaging Technology

Overview

There are many different people and organizations trying to use Globus (and succeeding). The Globus project strives to satisfy the needs of its users, but the reality is that different communities have different needs. Ideally, a tailored distribution could be constructed to meet the specific needs of an individual community. Right now, Globus' ability to be easily customized is minimal, and difficult to work with, at best. The goal of the packaging effort within the EMERGE project was to construct a framework and a set of Globus packages, which could be used to create, tailored Globus distributions that fit the specific needs of the organizations using Globus. Another goal of the packaging effort is to assist the process of developing the Globus toolkit, by allowing individual pieces of what is now a monolithic Globus distribution to be released on separate schedules.

The current monolithic Globus toolkit, which consists of many components, is inflexible in that it is difficult to distribute subsets of Globus. To alleviate this problem the Globus components will be divided into packages. Using the new packaging framework, it will be possible for organizations to construct both source and binary distributions of the selected packages they are interested in. No longer will users be forced to build and configure components, which are of no interest to them.

Under the new packaging framework it is possible to release updates to an individual component without necessarily releasing updates to all other components. Additionally, an individual component can be built and tested without configuring and building unrelated components. Thus the packaging framework substantially increases the efficiency of the development and release process.

This paper documents the design of the packaging framework, GPT, we developed and the specific application of GPT on the Globus Toolkit components.

Packaging Concepts

GPT provides a simple portable framework for creating and managing complex software packages. Here is a list of the features that are supported.

Install / Uninstall / Upgrade

The installation management of binary packages is the responsibility of a tool (or tools, collectively) known as the Package Manger. The Package Manager is the interface through which the person installing a package will install, uninstall, or upgrade it. An example of a package manager that is probably known to many of our readers is RPM, the RedHat Package Manager. On a RedHat Linux system, one typically uses RPM to install, uninstall, or upgrade any binary package.

Dependency tracking

There are three types of inter-package dependencies:

- **Compile dependencies** A compile (or compile-time) dependency exists when a package requires another package in order to compile. This usually means that the dependent package is including a header from the package on which it depends.
- **Link dependencies** A link (or link-time) dependency exists when a program or library in a package requires a library from another package in order to link. In the case of programs linked with shared libraries, the packaging system has to verify that the link dependencies have been fulfilled in a given installation, as otherwise the programs will not run.
- **Runtime dependencies** A runtime dependency exists when a program, script, or library requires some resource from another package at runtime.

Versioning

Whenever more than one separately released pieces of software need to interact, a need arises to ensure that the software, as installed, is interoperable. The time-tested way of doing this is to assign a version number to each release of each piece of software, so that any given version is readily identifiable. This allows the encoding of version numbers into package dependencies, as you may know that your package foo 2.11 requires package bar 2.0 or greater.

Flavored binaries

Certain compile time options used when creating a library, must also be used when a program is linked that uses that library. Otherwise, linking errors will occur. For example, it can be important that the same compiler be used, or that the same threads package be used. Additionally, it is very important whether or not we are compiling 32 bit or 64 bit. We refer to such sets of compile time options as flavors.

Relocatable binaries

It is important for ease of installation that the binaries not be tied to specific directories. (i.e. a program should not insist on being installed in /usr/local/bin). However, given large sets of inter-package dependencies, especially with scripts calling programs, it is a relatively hard problem to enable the easy installation of packages into totally arbitrary

locations. A reasonable compromise, which we make for Globus Toolkit packaging, is to insist that all packages comprising a given installation be installed into a single installation tree, but that tree can be rooted anywhere it is desired.

External programs and libraries

When distributing any software onto systems where the underlying operating system is not packaged using the same packaging system (i.e. every system onto which Globus is installed, unless someone makes a Linux or BSD distribution using our packaging system sometime in the future), there will be programs and libraries that do not have packaging system metadata associated with them. There are, in general, two ways of dealing with such external programs and libraries. One is to make special exceptions for them in the tools that check dependencies. The other, which is by no means mutually exclusive with the first, is to create "virtual packages" which consist of appropriate metadata for the external "package", and, if necessary, links from the install tree to the actual location of the external programs and/or libraries.

Compatible with existing package managers

The packaging system described in this document will have its own accompanying package manager, but the system has been designed with compatibility in mind. That is, it should be relatively simple to take a set of binary packages created for this system, and convert them into binary RPMS, for example. We are providing our own package manager so that we can ensure that a package manager is available on all platforms that we support, but organizations that will be creating distributions of Globus packages may wish to use their own package manager, for ease of installation management.

(A distribution of Globus packages is a set of Globus packages, along with the runtime configuration files (or tools to generate the runtime configuration files) needed for a particular set of users. RedHat linux is a good example of the distinction between packages and a distribution; each piece of software is a package, in an RPM, but the RedHat Linux Distribution includes tools (such as their install tool/bootdisk) that sets up the system for a particular configuration.)

Runtime Configuration files (vs. static data files) and who manages them (RPM vs. Linuxconf)

Programs, scripts, and possibly libraries, may require some information provided to them at runtime, per machine or per user, in order to function as desired. We will refer to files containing such information as runtime configuration files (we will always use this term instead of simply using "configuration files" to avoid confusing runtime configuration files with files used by configure when building a package). In this packaging system, binary packages may require that some runtime configuration files exist in order to function, but the package itself shall not install the actual files. This allows organizations that wish to create personalized distributions to create runtime configuration packages that can be installed and managed separately from the packages they relate to. This makes the process of upgrading a package without changing its runtime configuration files extremely easy--it is the default.

We can look to the Linux world for an example of this division. Linuxconf is a wizard that allows the user to relatively easily manipulate the runtime configuration files for various different packages that might be installed on a linux system. However, since RPM allows packages to install and manipulate their own runtime configuration files, there is no consistent method for ensuring that you retain your old runtime configuration when upgrading a package.

To illustrate this say that you have a globus package foo which has a runtime configuration file "foo.conf" that is modified by the user using the Linuxconf like GUI wizard "gui_fee". For this illustration the concept ownership is defined in such a way that when a package is responsible for installing, uninstalling, or updating a file it "owns" the file.

If foo.conf is owned by package foo then several problems arise. First any modifications by the user for gui_fee are lost whenever package foo is uninstalled, reinstalled, or updated. Second, foo.conf will be re-installed every time a new version of package foo is released even though the format of foo.conf most likely did not change. Some packagers have tried to resolve these problems by introducing pre and post install/uninstall scripts that are run during an action on package foo but this introduces an unacceptable amount of complexity to our packaging framework design.

The same problems occur when foo.conf is owned by the package gui_fee. In addition, gui_fee probably manages the runtime configurations of several packages not all of which have to be installed. Finally not all globus installations will be able to run a GUI wizard but will still need to have foo.conf.

The only acceptable solution is to have foo.conf in its own package freeing it from the actions needed for the other packages.

Package Types

Source Package

This package consists of source code, scripts, and documents, which are configured and built to produce binary packages. One source package will produce one or more binary packages each of which is a different package type. Source packages have two sources of dependencies to consider. The first source are the compile and link dependencies that are present when the source code is being built. The second source is the run-time dependencies that need to be stored in the binary packages when they are generated.

Source packages are different from all of the other package types, in that the package manager does not manage them. Source packages are not installed into the installation tree, so they do not need to include metadata for the purpose of their own removal. Rather, the metadata included in a source package is necessary for ensuring that the compile and link dependencies are satisfied when building the binary packages, and for generating the metadata necessary for each binary package being produced. A convenience tool which builds/installs/ generates binary packages from multiple source package ordered by their dependencies can also use the metadata.

Source packages shall have the following metadata:

- Name of the source package.
- Aging version of the source package.
- Types of binary packages produced.
- A flag indicating whether the package is built with flavors or not.
- A version specification of a configuration specification package if the package requires run-time configuration files.
- Compile Dependencies: Packages needed for compilation (headers etc.). Each dependency comes with a list of version specifications
- Linking Dependencies: Packages containing the libraries needed for linking. Note that this dependency ties to two different binary package types dev and rtl because the user will have a choice on whether to link against the shared or static binary of a package's library. Each dependency comes with a list of version specifications
- Build Env. CFLAGS line containing defines needed to use the header files. LIBS line containing external libraries need to link with the libraries in this package. And various other build flags.
- Runtime Dependencies: Packages containing programs, scripts, and data files needed to run programs or scripts in this package. These dependencies will be transferred to the binary packages that are built from this source packages. Each dependency comes with a list of version specifications

Dynamically Linked Program Binary Package (pgm)
This package contains dynamically linked executables and scripts. It will always be generated from a source package and will share the source package's name and version. If the package contains executables it shall also have a flavor as part of its identity. If the package contains only scripts then it can be designated as "noflavor".

Program packages can have run-time dependencies if their executables and scripts call executables and scripts in other program packages. They could also have runtime dependencies on data files and documents.

A program package can also have runtime linking dependencies if its executables are linked with libraries from rtl packages. For example, say that an executable links with libfoo in package fum. If the executable is linked to the shared library libfoo.so then the linking dependency translates to the fum_rtl package, which will have to be installed before the program package.

Dynamically linked program binary package metadata:

- Name of the package.
- Aging version of the package.
- Package type
- Flavor the package was built with (or noflavor). All of the libraries listed in the link dependency list will have the same flavor.
- A version specification of a configuration specification package if the package requires run-time configuration files.
- Runtime dependencies (including a version specification for each dependency) with other pgm, data, and doc packages.

- Runtime linking dependencies (including a version specification for each dependency) to rtl packages if the executables were linked with shared libraries.

Statically Linked Program Binary Package (pgm_static)
This package contains statically linked executables. It will always be generated from a source package and will share the source package's name and version. The package shall also have a flavor as part of its identity.

Static program packages can have run-time dependencies if their executables call executables and scripts in other program packages. They could also have runtime dependencies on data files and documents. In addition these packages absorb the runtime dependencies of the static libraries they are linked with. For example, consider a program foo that statically links with a library libfee.a that has a system call to still another program fum. The library libfee has a runtime dependency to the program fum. The program foo will have to absorb this dependency so that program fum is installed before program foo is installed.

A program package can also have regeneration dependencies if its executables are linked with libraries from other packages. For example, say that an executable links with libfoo in package fum. If the library was linked statically to libfoo.a then the dependency is translated to the fum_dev package. In this case the program package will have to be regenerated any time fum_dev is updated. A build number will be updated to reflect the regeneration.

None of the executables in a program package shall ever be built with a mixture of static and shared package libraries because this complicates the compatibility checks needed at runtime to make sure that all of the libraries are compatible.

Statically linked program binary package metadata:

- Name of the package.
- Aging version of the package.
- Package type
- Flavor the package was built with. All of the libraries listed in the package regeneration dependency list will have the same flavor.
- A version specification of a configuration specification package if the package requires run-time configuration files.
- Package regeneration dependencies (including a version specification for each dependency) to dev packages if the executables were linked with static libraries. This list is the same as the runtime linking dependencies list of the pgm packages except for the package type.
- Runtime dependencies (including a version specification for each dependency) with other pgm, data, and doc packages. In addition to the runtime dependencies of the executables and scripts, this list also contains the runtime dependencies of the static libraries for the entire regeneration dependency tree.

Development Binary Package (dev)

This package contains flavored header files, static libraries, and libtool library files. It will always be generated from a source package and will share the source package's name and version. The package shall always have a flavor as part of its identity.

Development packages can have run-time dependencies if their libraries call executables and scripts in other program packages. They could also have runtime dependencies on data files and documents.

Even though development packages are not installed for run-time they can still have run-time dependencies with other pgm, data, and doc packages if the libraries access files or programs in these packages. The run-time dependencies of a static library will have to be absorbed by a pgm_static package if it contains an executable that was linked with the library.

A development package can have a compile dependency to another dev package if it contains a header file that includes headers from the other package.

A development package can also have linking dependencies if its libraries use symbols from libraries contained in other packages. These dependencies are contained here so that the dependency tree for an executable (from a pgm_static) can be recursively extracted when the executable is built.

Development binary package metadata:

- Name of the package.
- Aging Version of the package.
- Package type
- Flavor the package was built with. All of the libraries listed in the link dependency list will have the same flavor.
- A version specification of a configuration specification package if the package requires run-time configuration files.
- Compile dependencies if files from the package include headers from other packages. In most cases this list is the same as the linking dependencies.
- Linking dependencies (including a version specification for each dependency) to other dev packages if its libraries use symbols from other libraries.
- Build Env. CFLAGS line containing defines needed to use the header files. LIBS line containing the libraries provided by this package, the external libraries needed to link with the libraries in this package, and various other build flags.
- Runtime dependencies (including a version specification for each dependency) with other pgm, data, and doc packages.

Non-Flavored Headers Package (hdr)

This package contains header files. It will always be generated from a source package and will share the source package's name and version. The package contains only header files, which are not configured for a flavor and so is assumed to be "noflavor".

A non-flavored headers package can have a compile dependency to another dev package if it contains a header file that includes headers from the other package.

Non-Flavored Headers binary package metadata:

- Name of the package.
- Aging version of the package.
- Package type
- Compile dependencies if header files from the package include headers from other packages.

Runtime Library Binary Package (rtl)
This package contains libraries used at run-time by programs and scripts. It will always be generated from a source package and will share the source package's name and version. If the package contains binaries it shall also have a flavor as part of its identity. Otherwise it is a noflavor.

Runtime packages can have run-time dependencies if their libraries call executables and scripts in other program packages. They could also have runtime dependencies on data files and documents.

Runtime packages have linking dependencies, which are needed at runtime. For example when a program using shared library foo starts execution, it needs to load libfoo.so as well as all of the shared libraries that libfoo depends on for symbols. Runtime library binary package metadata:

- Name of the package.
- Aging version of the package.
- Package type
- Flavor the package was built with. All of the libraries listed in the link dependency list will have the same flavor.
- A version specification of a configuration specification package if the package requires run-time configuration files.
- Linking dependencies (including a version specification for each dependency) to other rtl packages if its libraries use symbols from other libraries.
- Runtime dependencies (including a version specification for each dependency) with other pgm, data, and doc packages.

Data Binary Package (data)
This package contains data files, which cannot be modified by users. It will always be generated from a source package and will share the source package's name and version. If the package shall also have a flavor as part of its identity if any data files are configured for flavored. Otherwise it will be noflavored.

Data packages have run-time dependencies if data files include files from other data packages.

Data binary package metadata:

- Name of the package.
- Aging version of the package.

- Package type
- Flavor the package was built with or noflavor.
- Runtime dependencies (including a version specification for each dependency) to other data packages if its data files include files from other dev packages.

Document Binary Package (doc)
This package contains documents. It will always be generated from a source package and will share the source package's name and version. It will always be noflavored.

Document packages have run-time dependencies if document files include files from other doc packages.

Document binary package metadata:

- Name of the package
- Version of the package
- Package type
- Flavor the package was built with or noflavor
- Runtime dependencies (including a version specification for each dependency) to other document packages if its document files include files from other dev packages

4. Conclusions

Under the shortened project scope NCSA made two valuable contributions that significantly lower the barrier of Grid software deployment. Those achievements include the:

- Development of the Grid Packaging Toolkit (GPT) and
- The application of GPT to the Globus Toolkit.

The Grid Packaging Toolkit provides packaging capabilities that meet the needs of cross platform, organization specific configuration and includes support for version control, flavored binary releases, relocatable binaries, dependency checking, run time configuration, and packaging of external programs and libraries. Under the GPT packaging framework it is possible to release updates to an individual component of a complex software package without necessarily releasing updates to all other components. Individual components can be built and tested without configuring and building unrelated components. Thus the packaging framework substantially increases the efficiency of the development and release process. GPT allows virtual organizations to build and configure customized grid software packages, giving them the ability to more tightly coordinate the deployment of these complex packages. GPT supports the packaging of both binary and source releases. GPT is compatible with platform specific package managers such as RPM.

Finally GPT was successfully applied to the Globus Toolkit Version 2.0, which is available now. This application of GPT to Globus dramatically demonstrates GPTs ability package complex real world software.

Since the conclusion of this project we have continued to advance GPT. A recent packaging workshop was held among Grid developers and both the GPT framework and GPT were chosen as the preferred packaging tool. In addition to the Globus Toolkit GPT is being actively used to package the middleware software release for the NSF Middleware Initiative, and the Alliance Grid in the Box effort.

5. Publications & Presentations

GPT software can be found at ftp://ftp.ncsa.uiuc.edu/aces/gpt/

Comparison of GPT to RPM white paper
http://www.ncsa.uiuc.edu/Divisions/ACES/GPT/compare_rpm.html

Location of GPT packaged Globus Toolkit
http://www.globus.org/gt2/install/beta-download.html

Presentation on GPT given January 3, 2002
http://www.ncsa.uiuc.edu/Divisions/ACES/GPT/Wisconsin_GPT_Talk.ppt