Dynamic Deployment of a Component Framework with the Ubiqis System

Steven Brandt[1] Gabrielle Allen[1,2] Matthew Eastman[1] Matthew Kemp[1] Erik Schnetter[1,3]

[1] Center for Computation & Technology, Louisiana State University, Baton Rouge, USA
[2] Department of Computer Science, Louisiana State University, Baton Rouge, USA
[3] Department of Physics & Astronomy, Louisiana State University, Baton Rouge, USA

Abstract

Component frameworks provide one strategy for the development and deployment of complex multiphysics applications, which are increasingly developed by collaborative, distributed, and diverse teams, and are deployed across multiple supercomputer architectures located at numerous sites. One longstanding issue is the management of components. The complexity involved in choosing, locating, compiling, verifying, and deploying codes consisting of hundreds of independent modules can be overwhelming for beginners, and is a huge burden on developers and users. In the case of Cactus, framework-specific tools have been developed to address some of these issues, but it still suffers from being somewhat ad-hoc, difficult to maintain, and lacking in capabilities.

This paper describes a dynamic framework for deploying applications that we are developing, called Ubiqis, which works automatically and on-demand. It is decentralized, requires no per-application configuration by administrative staff, and can work recursively to resolve dependencies.

1 Introduction

Many large software projects use a component structure and depend on community collaboration to develop and maintain a diverse set of components. The resulting collection of components are maintained in different ways, e.g. through a single centralized source code repository, or for larger projects distributed repositories may exist where the collection of modules is maintained in a best effort manner through descriptions on web pages or by word of mouth, and information can easily be out of date or incomplete.

The Cactus framework is one such large project that involves finding, assembling and then compiling some hundreds of components from various locations on a diverse set of machines to solve problems in science and engineering. The issues involved in both assembling these components and also locating the appropriate libraries for compilation on parallel supercomputers is a major hindrance to using Cactus, particularly for new users. The Cactus user community has grown sufficiently that local ad-hoc methods of coordinating and tracking components are no longer adequate. In particular, providing a centralized component management mechanism is not feasible since the community consists of independent and competitive research groups that will not tolerate any central control.

In this paper, we describe a new dynamic framework for distributed component selection and deployment called Ubiqis. Ubiqis extends the Java naming convention and uses the capabilities of the FUSE file system to provide an automated and automatic system. Section 2 looks at the Cactus project and derives a set of requirements from its use cases. The major design points of the Ubiqis framework are then described in Section 3, and Section 4 describes the Cactus use cases that are addressed with Ubiqis, and evaluates Ubiqis' effectiveness for component management.

2 Case Study: Cactus Framework

As a case study for a system such as Ubiqis, we look at an example from a research community that uses codes based on the Cactus framework [7, 1]. The name Cactus is a metaphor for its design: The *flesh* provides a central set of infrastructure and interfaces upon which programmers can build and integrate components called *thorns*. Thorns can provide code for mesh generation, adaptive mesh refinement, I/O, checkpointing, generation of graphics, web servers, fluid dynamics, black hole physics, etc.

The Cactus flesh and a set of core thorns are developed and distributed via a CVS source code repository hosted at Louisiana State University. Research groups using Cactus typically use the core thorns, develop their own sets of thorns, and additionally take advantage of public or private thorns developed in various other groups and deployed on diverse servers. Groups usually use a variety of different source code repositories and repository management systems, including CVS, svn, darcs, and git. Researchers who use Cactus can be roughly divided into two categories, *de-*

velopers who write and maintain source code and commit new code into repositories, and *users* who download thorn sets to compile and run to solve specific problems in science and engineering while making only small changes to the code.

In the field of numerical relativity, more than 15 research groups have adopted Cactus as the underlying parallel framework and community toolkit. One sample group is found at the Center for Computation & Technology at Louisiana State University. This team of around ten researchers working on evolving black holes and neutron stars typically have simulation codes that are comprised of around 300 modules, taken from 20 different repositories. This group uses about 20 different supercomputers located at different institutions and via state-wide or national resource providers such as LONI [2] and the TeraGrid [3].

The Cactus configuration mechanism allows thorns to declare their dependence on a particular component interface (i.e., it *inherit* from it) or their ability to *implement* that interface. In a like manner, they can declare that they *require* or *provide* named functions or other capabilities. Generally, dependence of one Cactus thorn on another is not direct, but only via interfaces.

The Cactus build process provides information to say that one or more of these dependencies are violated. However, there is no satisfactory searchable database to facilitate finding thorns that will satisfy these dependencies. Instead, knowledge of which thorns implement what interface or provide a given function or capability is usually stored as a research group's cumulative experience, or found by inquiring on the Cactus users mail list. Cactus provides two mechanisms towards the distribution of thorn components:

GetCactus: This utility script takes as input a *thornlist* (list of thorn names), which can be augmented by location and authorization information for different CVS or SVN source code repositories. This is the usual way that Cactus users currently checkout their code and modules. Although the Cactus web site distributed a few standard thornlists, in general it is very difficult for users (and in particular new users) to construct a thornlist, and once a thornlist is constructed, it does not provide any opportunity to discover other thorns that provide the same capability.

MakeThornList: A second script takes as input a Cactus parameter file and uses the *ActiveThorns* parameter (that tells Cactus which thorns to activate for a particular run) to attempt to construct a thornlist. To do this, thorn names from the parameter file are compared with those in a *Master Thorn List* that contains a complete list of all thorns available to a particular user or group. This mechanism again does not address the issue of choosing between compatible thorns, and requires the administration and updating of multiple thornlists in an ad-hoc manner.

Figure 1 illustrates the typical workflow for users deploying Cactus, first for testing on their local workstation, and then for use on production supercomputing systems. Beginning with defining the science goals and designing a corresponding parameter file, there are usually several iterations of locating missing thorns, installing missing libraries, etc.

2.1 Cactus Use Case Scenarios

Here we outline three component selection and installation scenarios for Cactus that are also relevant for other component frameworks:

Publish Components: A research group wants to make its set of thorns available for use by the broader Cactus community, and needs a mechanism to publish and maintain an up-to-date description of thorns, and to make them available for download. Cactus users need a simple and reliable mechanism to search across all available thorns for components that implement a particular interface and then download chosen thorns.

Assemble Components Based on an Existing Parameter File: A user starts from a Cactus parameter file, perhaps passed on by a colleague from a different research group. The parameter file contains a list of the names of thorns to instantiate for a run. There should be a simple way to discover and assemble thorns compatible with a particular parameter file, ideally providing choices for alternative thorn implementations, checking parameter file consistency, and potentially listing additional thorns that may be added in to provide additional capabilities. (For example, different thorns provide different I/O or analysis methods that could be applied to a particular science problem.)

Assemble Components Based on Abstract Science Description: One of the key difficulties of using Cactus is that given a particular science problem, e.g. solve the scalar wave equation, or model the collision of two black holes, it is difficult to determine the list of thorns needed to build an executable (and to construct the parameter file for a simulation.) Fully solving this issue is an interesting problem which requires the introduction of more descriptive semantic information into the thorn interfaces related to the scientific capabilities and compatibilities. As a starting point towards this, our last scenario looks at assembling a compatible thorn list for a full simulation given just a few key thorns, for example an initial data thorn for scalar waves and one for the HDF5 I/O method.

2.2 Requirements from Case Study

The case studies for the Cactus Framework provide the following core requirements for a system to manage the selecting and assembling components. The system should:

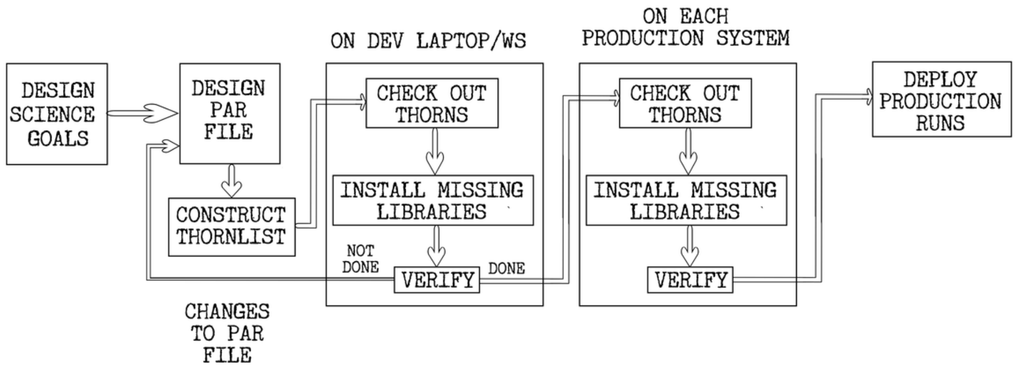

Figure 1. Typical workflow for solving an application science problem with the Cactus framework. After defining particular science goals, several iterations on local and remote computing resources refine the problem description in the parameter file until production simulations are launched.

- make it possible for users to update components to the most recent version (and also to choose not to update them);
- protect against outages of individual repositories, and automatic preservation of older versions of components;
- be trustable, i.e., a user must be confident they have really compiled and are running the expected code;
- allow for searching for different options for a particular thorn implementation, or the latest version of a thorn;
- minimize administrative burden on programmers and users alike – ideally, management should be decentralized;
- provide authentication to permit limiting access to non-public code.

3 Ubiqis: Design and Implementation

Ubiqis is a system for locating software on the web, loading and installing it on a local file system, and automatically resolving its dependencies. There is no central repository, code can come from anywhere on the web, and as the code is used, it is indexed, cached, and made available to a peer-to-peer network.

In this section, we describe the core architecture features of Ubiqis and their implementation, and we show how this relatively simple framework helps us to address the requirements defined in the case study.

3.1 Automatic Component Location

A basic concern identified from the use cases is to be able to find components distributed on the web. One solution to this challenge is inspired by Java language convention of naming packages according to the website of the publisher (e.g. `com.somecompany.somepackage`). This simple convention offers a starting point for locating software. In Ubiqis, this is extended to a more formal convention that completely identifies the published code.

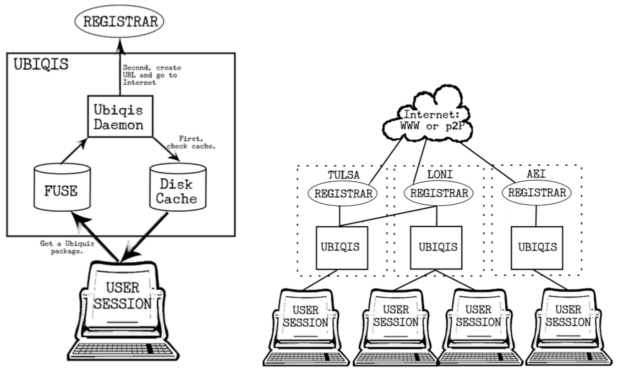

Figure 2. [Left] Ubiqis uses FUSE to make software available on-demand, either fetching from Cache or the internet. [Right] Ubiqis registrars provide proxy access to the internet, caching and sharing data in a p2p network. Multiple users share a single Ubiqis install, and Ubiqis installs can talk to many registrars.

To simplify integration with makefiles, compilers, etc., FUSE (File System in User Space) is used to interface to the web. FUSE intercepts calls to a virtual file system, translates Ubiqis names in the path, then downloads and caches files as they are used (See Figure 2).

The following example shows how name resolution can be coupled with FUSE to automate the finding and deployment of a header file. In the example, `/mnt/ubiqis` is the root of the FUSE file system.

```
$ cat test.cpp
#include "edu/lsu/ubiq2/repo/thecode/inc/
   header.h"
  ...
$ g++ -I /mnt/ubiqis test.cpp
```

When g++ attempts to load `header.h` from the `/mnt/ubiqis` file system, the Ubiqis daemon recognizes the set of path elements as pieces of an Ubiqis name. It constructs the URL and fetches the package containing the header. If that header contains other Ubiqis includes, Ubiqis will detect them also, and transparently load the dependency. Ubiqis effectively enables g++ to download code from the internet as needed – although g++ was not instrumented and Ubiqis has no special knowledge of g++.

This illustrates a feature of the Ubiqis system that could be used to significantly extend the functionality in Cactus – code can be deployed as collections of header files separate from the Thorn mechanism. The headers would then be transparently deployed and shared between thorns without any particular awareness on the part of a Cactus user that it was happening.

The significance of this ability to distribute header files inside code is not to be underestimated – a large fraction of the intricate C++ Boost libraries are distributed this way [10]. While Cactus does not use C or C++ exclusively, a significant set of its most important thorns are based on it.

3.2 Searchability and Resiliency

The Ubiqis system as described so far is vulnerable to outages. If a website publishing a component is sold, reorganized, or experiences down time, access to the software component it distributes will be lost. We want to protect against this situation, but without causing administration and centralization issues.

To address this, Ubiqis uses registrars – web services that act as proxies for Ubiqis requests. The registrars fetch metadata objects on behalf of a user, sign them and return them. Thereafter, a registrar should never contact the publisher website for meta data again; it should simply serve the meta data file from cache.

Thus, if a publishing website or one of the registrars goes down, the data is still accessible. Because the package metadata carries the signature of the full package, it is possible to obtain a copy of it from elsewhere and verify that it matches the metadata file.

Another Ubiqis feature is searchability, obtained by simply connecting the registrars together in a peer-to-peer network and allowing the metadata files to contain search keys.

3.3 Versioning

To work correctly, a package should never change once it has been released through Ubiqis. For this reason, it makes sense to make the version number of the software part of the name. We propose that the version be encoded in a special element which is easy for the registrar to sort.

Thus, unlike other keys into the Ubiqis network, the one with keyword "latest" can be used to serve dynamic content to the network of Ubiqis users. Combining this with the fact that the peer-to-peer network can protect access to the component in the face of server outages, the URL becomes more like an abstract handle, similar in many ways to that employed by the Handle System [11].

4 Cactus Distribution and Assembly

Here we describe how Ubiqis has been applied to the three component assembly use cases for Cactus discussed in Section 2.1. The first scenario involved discovering components developed by other groups. The thorn developer ensures that his Ubiqis configuration is using a registrar that points to some element of the peer-to-peer network. The developer then publishes the thorn on a website under his or her control, and does a build to test whether it was done correctly (no errors in path name, sha1 sum, etc.). Assuming no errors occur, the thorn is automatically indexed and cached by the registrar and offered to the network at this point. If this thorn has a newer and higher version number than an existing version, the registrar network will update its notion of the latest version of this thorn. From now on, any attempts to search for the thorn or its keys will succeed.

In the second scenario, assembling thorns to run a particular parameter file, we simply supply a list of fully qualified Ubiqis names in the Active Thorns section of the parameter file (e.g., for `IDScalarWave` the full Ubiqis name is used `edu::lsu::cct::ubiq4::_tsbrandt::v1::CactusWave::IDScalarWave`). Minor changes have been made to the Cactus build system so that, starting from a parameter file and no locally installed thorns, Cactus parses the Ubiqis name in the parameter file. As the Cactus make system encounters thorns and attempts to read their source files, Ubiqis discovers, downloads, and installs them.

In the third scenario, we start from an abstract science description, basically just two thorns:

```
edu::lsu::cct::ubiq4::_tsbrandt::v1::
   CactusWave::IDScalarWave
edu::lsu::cct::ubiq4::_tsbrandt::v1::
   CactusPUGHIO::IOHDF5
```

The `IDScalarWave` thorn provides initial data for a simple evolution of plane waves in three dimensions. The `IOHDF5`

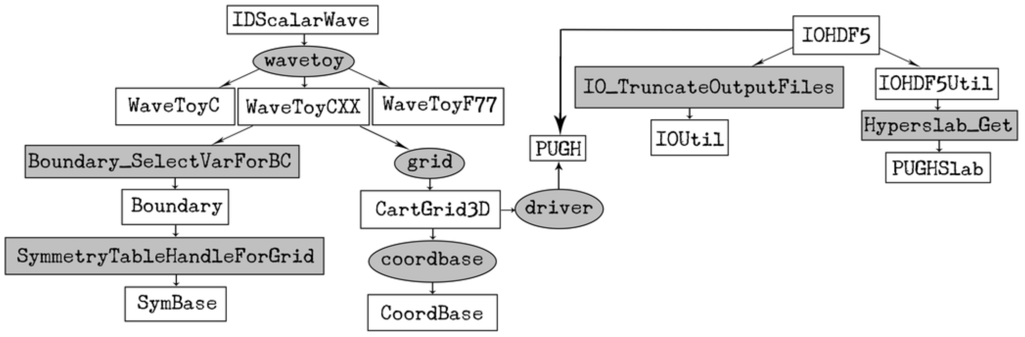

Figure 3. Outline of dependencies between Cactus thorns for testing the third use case. Gray ovals represent interfaces, gray squares represent abstract functions, white squares represent thorns.

thorn provides a simple interface to generating HDF5 output files. Attempting to build this thorn causes both the named thorns to load into the Ubiqis repository (if they are not already cached.) The build then produces an error, namely that a thorn implementing "wavetoy" has not been provided for `IDScalarWave`. A search is sent to the Ubiqis registrar, and three results are returned:

```
$ ubifind 'cctk:implements wavetoy'
http://cct.lsu.edu/~sbrandt/v1/CactusWave/
   WaveToyC.ubiq
http://cct.lsu.edu/~sbrandt/v1/CactusWave/
   WaveToyCXX.ubiq
http://cct.lsu.edu/~sbrandt/v1_41/CactusWave/
   WaveToyC.ubiq
```

Since there are three thorns implementing this interface, one needs to be selected. We choose `WaveToyCXX` (an algorithm could select the most popular thorn, the thorn with the best performance on a machine, etc.,) add it to the ThornList, and attempt to build again. This procedure iterates until we have a complete list, eleven thorns, and the compilation completes successfully. The dependency path traced by this process is shown in Figure 3.

It should be empahsized that the above search was made using keywords embedded by the developer in the Ubiqis meta data file, and that Ubiqis has no special knowledge of Cactus or its thorns.

This proof-of-concept test case was repeated using different implementations of the above thorns and with alternative IO packages, correctly locating and building components as needed.

5 Related Work

There are a large number of component repositories and systems for managing them, developed by various communities of developers. A major task that they handle is automatic dependency resolution. Generally, this is accomplished by having an explicit, per package, dependency list that is explicitly declared and is processed by some central server, or list of servers, for resolution. Frequently, these central servers require some kind of license to be agreed to, and this may or may not pose a problem for the community.

One area where the need for dynamic deployment is readily recognized is in Grids. As of 2006, a standard called ACS (Application Contents Service) has been in place to address this problem automatically, and NAREGI [8] is the prominent implementation of that standard. ACS is based on a concept of a trusted repository (Application Archive) where code can be stored for distribution.

NAREGI targets a similar problem space to Ubiqis, but has a broader reaching grid focus. The NAREGI component identified as WP-6 targets dynamic installation of application-specific grid middleware, and it is this that we focus on. NAREGI provides an interface where users can register, deploy, compile, and retrieve their grid applications using a Java applet running inside a browser. A significant amount of knowledge is stored by NAREGI to assist in builds on diverse architectures.

The EDOS system, like Red Hat's RPM and Yum, is designed for distributing Linux components (Mandriva, in particular) and is based on the model of a small set of publishers distributing content to a large user network. One unique feature of the EDOS/Mandriva system is that it uses a peer-to-peer distribution system for content deployment [5].

The Java community can benefit from one of the more advanced component systems built to date: Maven [9]. It is cited as being one of the more relevant higher-level build management techniques in CBSE [6], and is widely used in Eclipse projects. Maven uses a central repository for most of its code distribution (although it is possible to build local repositories as well). Each Maven project creates a POM (Project Object Module) in which dependencies of the project are explicitly declared. Projects with correctly constructed POMs will be able to automatically resolve and download dependencies into a special per-user cache, then complete a build automatically.

Ubiqis, in contrast to the above, discovers dependencies implicitly, only loading them as they are needed. Instead of a specialized repository, it can simply use any web server on any site to offer packages. Registrars are optional (but highly desirable) and exist to facilitate searching, fault tolerance, and security. Registrars, like the Ubiqis daemon itself, do much of their work implicitly.

6 Future Work

While this paper has illustrated a proof-of concept demonstration of the Ubiqis system, more work is needed to complete the feature set and to prepare for production use. Some of the key areas of immediate Ubiqis future work involve peer-to-peer capabilities, mutation, and access authorization as described below.

One of the key features of the Ubiqis system is the peer-to-peer network. While it is essential for the production release of the Ubiqis system, it is not necessary for demonstration of basic functionality. The incorporation of this technology should be straightforward, as it does not require any particular innovation, and an off-the-shelf implementation such as Pastry [4] is a likely choice.

We have discussed the need to limit access to some repositories in the Ubiqis network. We believe that the http basic auth mechanism, combined with a basic password file (similar to `~/.cvspass` used by CVS) can be used to limit access both during installation and invocation from the cache. In this case the relevant search keys could be limited to registrars configured to use them.

Our implemented Cactus use case scenarios have shown how advanced tools can be built around the existing Cactus interface descriptions to facilitate code assembly. This points the way towards future enhancements of a richer description language for the science and computational capabilities of thorns that can fully automate the selection of the best available thorn list for a particular problem. For example, tags might include performance or memory use metrics, and search keys might make use of server data to enable discovery of the most popular (widely installed) thorns. If coupled with a more generic search interface then users should eventually be able to request a set of thorns for "evolving initial data for two orbiting black holes using adaptive mesh refinement, finite differencing, and the BSSN evolution method, calculating gravitational waveforms, and outputting fields in the HDF5 data format."

7 Conclusions

We have described the Ubiqis system for management, deployment, and use of components and shown its use with the Cactus Framework as an example of a more general component software distribution. Ubiqis minimizes the need for administrative oversight, or centralization, and removes the need for special repositories or repository software. This is achieved by establishing a naming convention for components that allows them to be converted algorithmically to the URLs from which the component code may be downloaded. Use of a FUSE file system means that access of components, by compilers, preprocessors, or other tools will automatically trigger loading and installation by simple access.

This automatic access can also work recursively. As a tool attempts to access dependencies (e.g. a C-preprocessor attempts to load nested includes or a Java interpreter or compiler loads package dependencies) the needed code is downloaded and installed, resolving dependencies on the fly without explicit identification by the component's author.

Since these downloads go through proxies, called registrars, that are networked together in a peer-to-peer network, normal use of a component (even by its author on his/her own machine) will cause released software to become registered and its search keys indexed for community use.

Acknowledgments

We would like to acknowledge the contributions of Tom Goodale, Ian Kelley, Frank Löffler, and Thomas Radke in developing the `GetCactus` tool, Nancy Brandt for providing graphics services, and Daniel S. Katz and Peter Diener for their helpful comments. This work was supported by the Center for Computation & Technology, and NSF awards Alpaca (#0721915) and XiRel (#0701566).

References

[1] Cactus computational toolkit. http://www.cactuscode.org/.

[2] Louisiana optical network initiative. http://www.loni.org/.

[3] NSF TeraGrid. http://www.teragrid.org/.

[4] Rice university's peer-to-peer substrate. http://freepastry.rice.edu.

[5] S. Abiteboul, I. Dar, R. Pop, G. Vasile, and D. Vodislav. Large scale p2p distribution of open-source software. *Very Large Database Conference*. Vienna Austria, Sep 2007.

[6] O. Dalle. The OSA project: an example of component based software engineering techniques applied to simulation. In *The 2007 Summer Computer Simulation Conference*, 2007.

[7] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science*, Berlin, 2003. Springer.

[8] H. U. H. Kanazawa and S. Kawata. *A Problem Solving Environment based on Grid Services: NAREGI-PSE*, volume 239. Springer Boston, 2007.

[9] V. Massol and T. O'Brien. *Maven: A Developer's Notebook*. O'Reilly, 2005.

[10] R. Ramey. Making a boost library. In D. Musser and J. Siek, editors, *Library-Centric Software Design*, 2005.

[11] S. X. Sun. Establishing persistent identity using the handle system. *Proceedings of the Tenth International World Wide Web Conference*. Hong Kong, May 2001.