Rolls: Modifying a Standard System Installer to Support User-Customizable Cluster Frontend Appliances

Greg Bruno, Mason J. Katz, Federico D. Sacerdoti and Philip M. Papadopoulos
The San Diego Supercomputer Center
University of California San Diego
La Jolla, CA 92093-0505
{bruno,mjk,fds,phil}@sdsc.edu
http://www.rocksclusters.org

November 9, 2005

Abstract

The Rocks toolkit [9], [7], [10] uses a graph-based framework to describe the configuration of all node types (termed *appliances*) that make up a complete cluster. With hundreds of deployed clusters, our turnkey systems approach has shown to be quite easily adapted to different hardware and logical node configurations. However, the Rocks architecture and implementation contains a significant asymmetry: the graph definition of all appliance types *except the initial frontend* can be modified and extended by the end-user before installation. However, frontends can be modified only afterward by hands-on system administration. To address this administrative discontinuity between nodes and frontends, we describe the design and implementation of *Rolls*. First and foremost, Rolls provide both the architecture and mechanisms that enable the end-user to incrementally and programmatically modify the graph description for *all* appliance types. New functionality can be added and any Rocks-supplied software component can be overwritten or removed simply by inserting the desired Roll CD(s) at installation time. This symmetric approach to cluster construction has allowed us to shrink the core of the Rocks implementation while increasing flexibility for the end-user. Rolls are optional, automatically configured, cluster-aware software systems. Current add-ons include: scheduling systems (SGE, PBS), Grid Support (based on NSF Middleware Initiative), Database Support (DB2), Condor, Integrity Checking (Tripwire) and the Intel Compiler. Community-specific Rolls can be and are developed by groups outside of the Rocks core development group.

1 Introduction

Commodity clusters have emerged as the dominant computational tool for high-performance computing. This claim is supported by the fact that 58% of the Top500 fastest supercomputers [4] in the world are commodity clusters – the next closest classification occupies 24% of the list. Yet, high-performance computing clusters are only the most visible instantiation of this class of machines. Clusters are becoming prevalent in other areas including databases, distributed (grid-based) systems, visualization, and storage. These types of clusters are differentiated from HPC by their inclusion of specialized hardware subsystems (e.g., high-end video card or a storage area network (SAN) interface) and/or the additional software packages and services that must be installed and configured.

A storage cluster, for example, usually doesn't need a batch scheduler, but does need a specific partitioning scheme and a properly configured parallel file system. On the other hand, both HPC and storage cluster builders may want to add grid capabilities for remote access. One read-

ily observes that clusters of all types generally have a significant amount of shared functionality and are differentiated by a relatively small number of software services.

Traditionally, the designers of open-source [7], [5], [15], [14], [1], [12] and commercially available cluster management offerings [2] have focused almost exclusively on HPC. Groups that need to build non-HPC clusters have been forced to adopt an HPC stack then manually customize the machine by both removing undesired functionality and adding missing functionality. Or worse, they opt to build their own complete solution. Unfortunately, many groups start over, build their own cluster system and re-invent significant functionality that could be leveraged from the HPC space.

We believe that the basic problem stems from a common construction technique (which is also used in Rocks); the frontend (or head node) is built first and then all cluster nodes are installed using services on the frontend. Because the frontend is the first component bootstrapped in a cluster, it occupies a unique role in the complete system configuration and is therefore the most difficult to fully customize. Most end-users have to rely on competent system administrators to make substantive changes to the frontend *after* it has been installed. In our earlier paper [9], we described cluster management as devolving into administering two systems; the frontend and the compute nodes. Unfortunately, significant hand configuration often leads to unintended errors, reducing overall reliability, and making system reproducibility difficult to achieve.

A key goal is to remove the administrative dichotomy between frontends and all other node types. In [7] we reported on our graph-based configuration method for building node appliances. Our complete system allows us to programmatically describe the configuration of nodes and automatically deploy them across relatively large clusters [10]. This paper extends that work and introduces *Rolls* which, at their core, allow us to create and modify a complete system graph before frontend build time, just like all other appliances. What this means is that users can systematically modify a frontend configuration *a priori* and not rely on a system administrator to make changes afterward. The frontend still serves as a bootstrap node for the entire cluster preserving this well-accepted division of labor.

From a user's perspective, she only needs to insert Roll CDs at frontend build time to extend (or overwrite) configured software services. Optional software such as batch schedulers, grid services, integrity checkers, and community-specific customizations are trivially added, and more importantly, configured correctly without active administration. The Roll itself, created by an expert and downloaded by the end-user, contains all necessary information. Specifically, how to graft on new subgraphs to the Rocks core, which packages are needed for various appliance types, and what new information (if any) needs to be gathered from the end-user to properly configure new services. Effectively, Rolls are first-class extensions to the base system.

This approach has several key benefits: 1) the Rocks core functionality is actually made smaller (and hence easier to maintain as a robust system) but is overall more flexible; 2) other clusters types are more easily defined and inherit the turnkey deployment mechanisms of the Rocks core; 3) the entire system is more reproducible because a large amount of administrative uncertainty is removed; and 4) external developers need to focus only on additional (or replacement of) functionality instead of building a completely new cluster stack.

Section 3 provides an overview of the Rocks Cluster Distribution (the software system that uses the method described in this paper). Section 4 describes Rolls in detail. Section 5 details the mechanisms used to graft Rolls into a standard system installation tool. Lastly, in Section 6 we provide a summary of our work.

2 Related Work

After any cluster is physically assembled, a user must define a complex software stack for all nodes and then physically install them. Common practice is to first install a *frontend* (alternatively a head-node); this machine houses user data as well as the tools to develop and run cluster-wide services such as a batch scheduler, domain name server, and cluster monitoring. Compute and other types of nodes are then installed using instructions or images housed on the frontend.

There are three approaches used to install cluster software services: manual, add-on and integrated.

2.1 Manual Approach

In the manual approach, a user with significant system administration experience brings up the frontend by installing the base OS (e.g., by using a installation utility bundled within a Linux distribution such as Red Hat or SuSE). After the base OS is installed, all cluster-specific software is downloaded and/or installed on the frontend. These extra packages are merged with the base OS to produce an augmented RPM-based distribution (or a "golden image") that will be used to install the compute nodes. Manual configuration and maintenance of DHCP and PXE services (or unique boot floppies/CDs) is used to assign unique names and IP address to compute nodes and to push the manually assembled software stack to the compute nodes.

2.2 Add-On Method

Similar to the manual approach, the *add-on* method also requires a system administrator to first install a base OS on the frontend. After the frontend is installed and booted, the cluster building components are downloaded and installed. For example, tools to uniquely name and install compute nodes are included within these packages that reduce the complexity of manipulating the DHCP and PXE services as described in the previous section.

Cluster toolkits such as OSCAR [5] and Warewulf [15] are representative of this add-on method. Furthermore, these toolkits focus exclusively on HPC and assume a highly-competent cluster-aware system administrator to make any substantial changes to either software configuration or installation of new cluster-wide services.

2.3 Integrated Method

In the integrated method, the base OS, the cluster-specific services, and cluster building tools are bundled into one distribution. All cluster services and tools are installed and configured during the initial installation of the frontend, that is, no outside packages need to be downloaded and manually configured in order to bring up an entire cluster.

Examples of this method are Scyld [2] and Rocks. Scyld uses a modified kernel and system libraries in order to support their job control features [6] while Rocks leverages the default kernel and libraries as supplied by Red Hat.

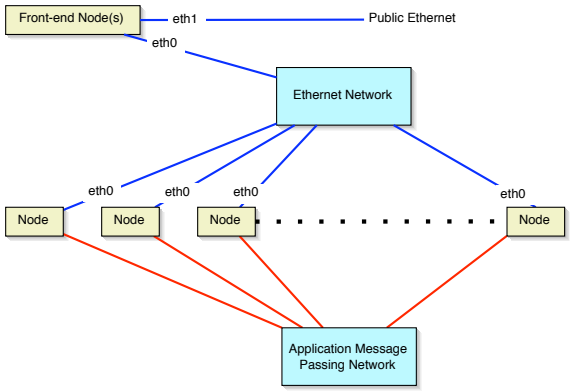

Figure 1: **Rocks hardware architecture.** Based on a minimal traditional cluster architecture.

3 Rocks Cluster Distribution Overview

Rocks is a cluster-aware Linux distribution based upon Red Hat with additional packages and programmed configuration to automate the deployment of high-performance Linux clusters[1]. Rocks is deployed on a minimal traditional cluster architecture (Figure 1) [13, 11]. This system is composed of standard high-volume servers, an Ethernet network and an optional off-the-shelf high-performance cluster interconnect (e.g., Myrinet). We have defined the Rocks cluster architecture to contain a minimal set of high-volume components in an effort to build reliable systems by reducing the component count and by using components with large mean-time-to-failure specifications.

Software installation within Rocks unifies two orthogonal processes: the installation of software packages and their configuration. The traditional single desktop approach to this process is to install software packages and then, through a process of manual data entry, configure the installed services to one's requirements. A common

[1] For a summary of other cluster-building tools, see [7].

extension of this process to clusters is to hand configure a single node and replicate this "golden image" onto all nodes in a cluster. Although this works with homogeneous hardware and static cluster functional requirements, we have found that clusters rarely contain either of these attributes.

Rocks treats software installation and software configuration as separate components of a single process. This means that manual configuration required to build a "golden image" is instead automated, and no image is ever built. The key advantage is the dissemination of intellectual property. Building cluster "golden images" is more often than not an exercise in replicating the work done by other cluster builders. By automating this configuration task out-of-band from system installation, the software configuration is specified only once for all the Rocks clusters deployed world-wide[2].

Installation of software packages is done in the form of package installs according to the functional role of a single cluster node. This is also true for the software configuration. Once both the software packages and software configuration are installed on a machine, we refer to the machine as an **appliance**. Rocks clusters often contain *Frontend*, *Compute*, and *NFS* appliances. A simple configuration graph (expressed in XML) allows cluster architects to define new appliance types and take full advantage of code re-use for software installation and configuration. Details of this framework are discussed in a previous paper [7], the Roll extensions are presented in Section 5.

The design goal of Rocks is to enable non-cluster experts to easily build and manage their own clusters. To support this goal, we identify two key design elements in Rocks: 1) the automation of administration functions, and 2) repeatability. We ensure everything developed in our lab can be easily deployed in other groups' labs. The first goal is critical for fully-automated node installation, an important property when integrating clusters at scale [10]. The second goal, the topic of this paper, is addressed by the construct we call Rolls.

[2]As of July 2004, Rocks has been deployed at over 200 sites representing over 51 teraflops of peak computing (`www.rocksclusters.org/rocks-register`).

/export/profiles/nodes/sge-client.xml
/export/profiles/nodes/sge-globus.xml
/export/profiles/nodes/sge-server.xml
/export/profiles/nodes/sge.xml
/export/profiles/graphs/default/sge.xml

Figure 3: **SGE Roll configuration package files represent the configuration sub-graph for SGE.**

4 Rolls

A Roll is a self-contained ISO image that holds packages and their configuration scripts. Figure 2 displays the contents of the SGE roll. This layout is nearly identical to the layout of a Red Hat produced ISO image. This was intentional as 1) it allows us to use Red Hat supplied scripts to assist in building the Roll, and 2) it is a familiar structure for those developers who have previous experience with building Red Hat distributions.

The `RedHat/base` directory contains two files that can be used by Red Hat's distribution utilities to query the contents of the Roll. Currently, our extensions to Red Hat's installer do not access these files (our extensions directly access the contents of the Roll in order to install packages). These files are present to support our experimentation regarding future Roll features.

The `RedHat/RPMS` directory contains the binary packages that comprise the Roll. One package in particular holds all the configuration information for all the packages found on the Roll. In the case of the SGE Roll, this package is named `roll-sge-kickstart-3.2.0-0.noarch.rpm`. Figure 3 displays the contents of this package with XML files in the `nodes` and `graphs/default` directory. The files in the `nodes` directory are vertices in a configuration graph, while the file in the `graphs/default` directory defines the edges that connect the SGE vertices to one another as well as the connections to vertices outside the SGE Roll. Each of the vertices from the SGE Roll sub-graph contains (expressed in XML) a list of Red Hat packages and optional configuration scripts to turn a meta-package into a fully configured software deployment. A deeper description of Roll vertices and edges is found in the next section.

To create a Roll, the developer must first checkout the

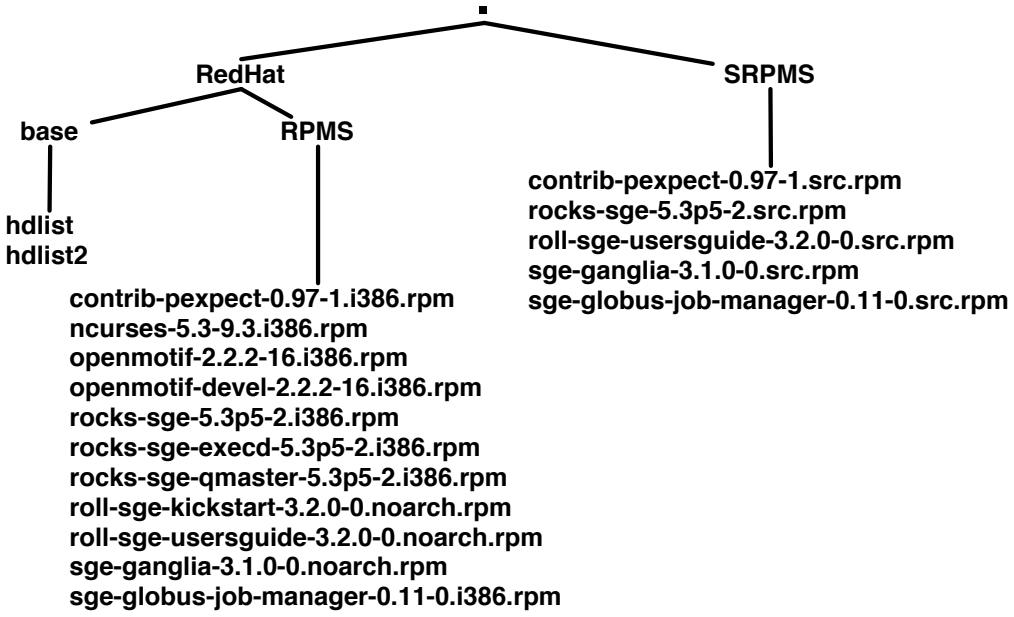

Figure 2: **Contents of the SGE Roll.**

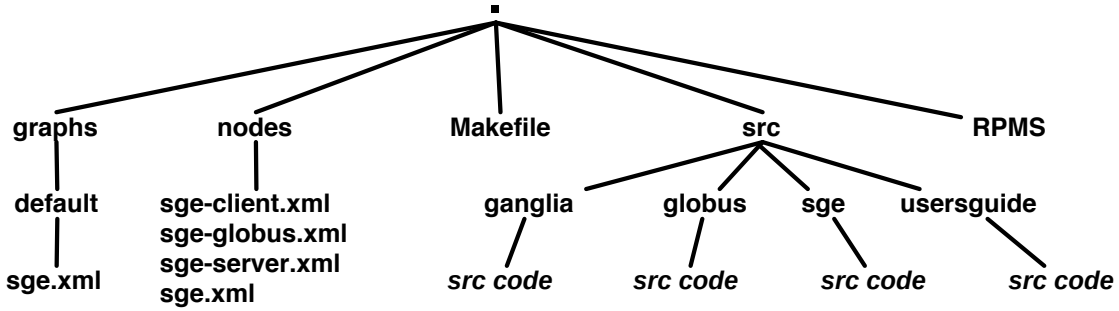

Figure 4: **Build directory for the SGE Roll.**

Rocks source tree and then create a directory structure similar to the SGE Roll as displayed in Figure 4. All source code is housed under the `src` directory. Binary RPMs that are created after the source is successfully built are stored in the `RPMS` directory. Additionally, any 3rd-party binary RPMS can be manually placed into the `RPMS` directory. The graphs and nodes files are the files that will comprise the configuration package (in the case of SGE, this is the package named `roll-sge-kickstart-3.2.0-0.noarch.rpm`). The Rocks build environment automatically constructs this package when the developer executes the command `make roll`.

The command `make roll` is used to create a Roll ISO image. The first task performed is to automatically construct the configuration package for the Roll. Then every directory under `src` is traversed resulting in newly built binary RPMS. Finally, an ISO image is built (resembling the form of Figure 2).

One of the extensions we applied to the standard Red Hat installer is to search for and apply the configuration package for each Roll to the installation environment. This allows Rolls to dynamically alter the distribution as seen by the standard system installer. How this functions is the topic of the next section.

5 Graph-Based Configuration

Prior to the introduction of Rolls all software installation and configuration was driven from a single configuration graph describing the construction of all machines in a cluster. This graph, the subject of [7], expressed both the set of packages to be installed and the individual package configuration. While the former is reasonably generic and applies to all clusters, the second is highly site-specific and defines what makes a cluster unique. This configuration graph was composed of nearly two hundred *vertex* files and a single *graph* XML file. Each *vertex* file specified the complete software package and configuration information for a specific function. For example, the single *vertex* file `ssh.xml` described the SSH service that permits user logins on all nodes in the cluster. The *graph* file specified the directed graph edges that connect the vertices to each other. Using this XML representation of a configuration graph complete software configuration for any cluster node type could be produced by traversing the configuration graph from the appropriate entry point *vertex*. For example, the configuration graph had vertices named **compute** and **frontend** which were the entry points for building compute nodes and frontends. Rolls further decompose this configuration from a single graph of hundreds of nodes, into multiple sub-graphs which are assembled at installation time into what was previously a single monolithic framework. Figure 5 is a representation of the core Rocks sub-graph which describes the configuration of appliances in a Rocks cluster, minus any Roll provided configuration.

This core configuration graph describes the minimum functionality to build a **server** and a **client**. Where a **server** is defined as a machine that has the ability to build other machines (e.g., the frontend), and a **client** is defined as a machine built by a **server** (e.g., compute nodes). The implementation of this is a Red Hat Kickstart file produced by the server node whenever a new client node is added to the cluster. This Kickstart file is compiled from a traversal of the graph starting at the **client** vertex. Similarly, the frontend node is also Kickstarted when the cluster is first constructed using this same mechanism, except the traversal begins at the **server** vertex. While this base configuration graph describes the different functionality between frontend nodes and compute nodes, it does not include a full description of standard cluster computing tools such as MPI, PVM, Ganglia, Globus, Sun Grid Engine, and other cluster middleware. This additional information is completely contained on the respective Rolls.

Figure 6 is the configuration graph for the SGE Roll. On the left-hand side of the figure, there are two uncolored vertices: **server** and **client**. These describe what the configuration of frontend SGE services and compute node SGE services are with shared functionality found within the **sge** vertex. The **server** and **client** vertices are uncolored because they are foreign to the SGE roll and are *well-known graft points* into the core Rocks configuration graph. When the SGE Roll is grafted into the core graph, the cluster's frontend and compute node configurations are updated to include the Sun Grid Engine service. This graph also includes another uncolored (and foreign) **globus** vertex which has a directed edge to the **sge-globus** vertex. The **sge-globus** vertex describes the configuration of SGE GRAM which allows Globus job submissions to run on the local scheduler, something that

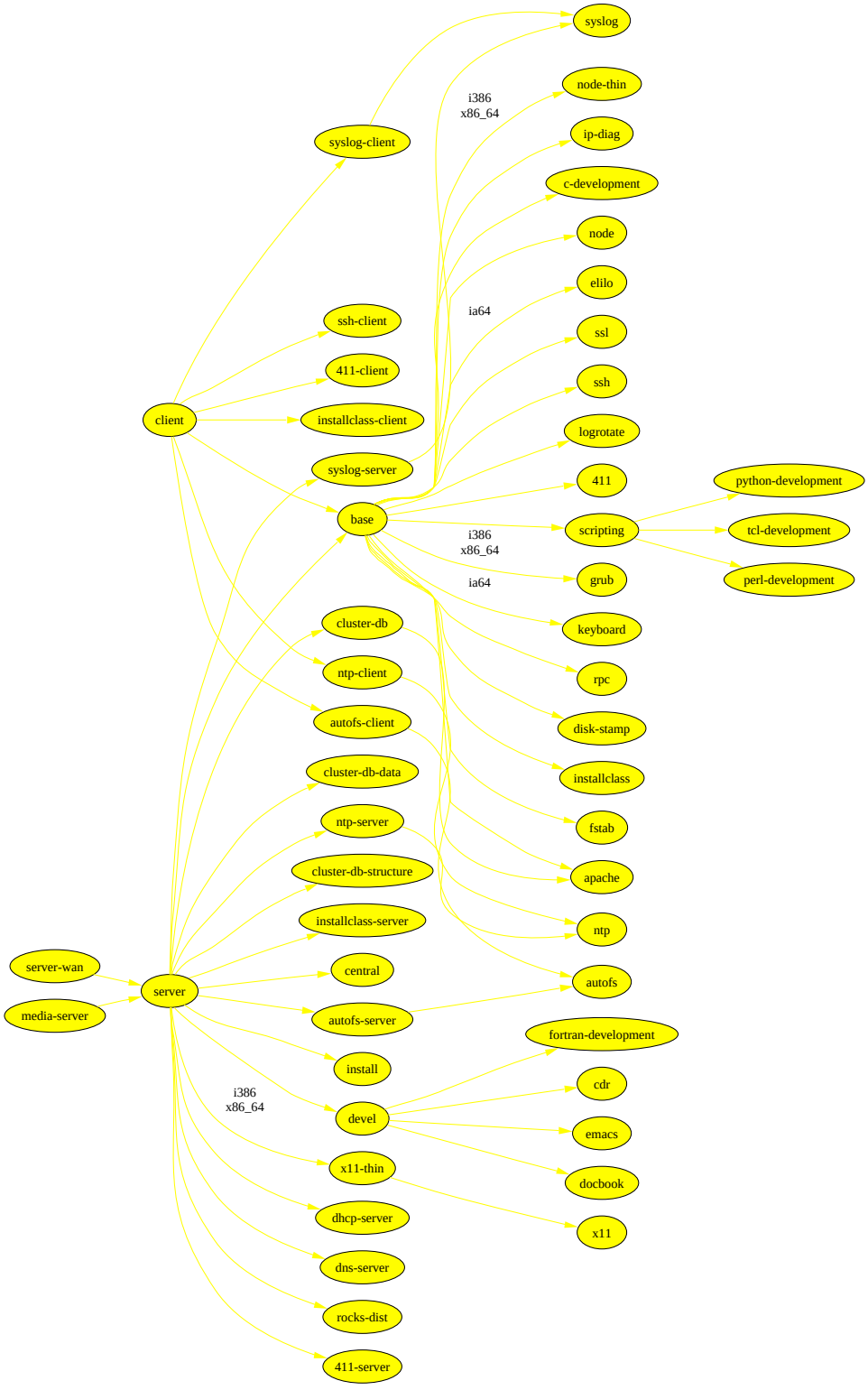

Figure 5: **Core Rocks Configuration Graph.**

is highly scheduler specific. However, Globus support is included in Rocks via the Grid Roll. By expressing this **globus** → **sge-globus** dependency in the SGE Roll, we can configure the SGE GRAM if and only if the Grid Roll is present during installation. In other words, without the Grid Roll included, the **globus** vertex will be absent from the fully-assembled composite graph and no path to **sge-globus** will be preset. In this manner, a Roll's sub-graph is not only grafted into the core configuration graph, but it may also include inter-roll relationships.

In addition to the optional SGE Roll, Rocks provides a required HPC Roll which also contains the foreign vertices **server** and **client**. The HPC graph describes the configuration of MPI, PVM, Ganglia, and other standard HPC software commonly deployed on a cluster. We view the HPC Roll as the minimum additional configuration required to build a complete MPI Cluster. Figure 7 is the composite graph of the core, HPC, and SGE sub-graphs. The color of each vertex and edge indicates the Roll it came from. Note that the core functionality is found primarily on the right-hand side of the graph (near the leaf nodes), the HPC configuration is found on the left-hand side (near the root nodes), and the SGE configuration is peppered throughout. This indicates that the HPC Roll builds on top of the core Rocks configuration, while the SGE Roll affects all aspects of configuration.

Figure 8 zooms in on the center of Figure 7. In this view of the composite graph, functionality and inter-roll relationships are expressed from the core and the HPC and SGE Rolls. In our previous monolithic framework, this information was expressed in a single central place, where now logical component pieces are identified as *sub-graphs* and engineered and maintained independently from one another. It is by our just-in-time graph assembly that this composite graph is produced prior to system installation that a complete Rocks core, HPC, and SGE configuration can be instantiated as a single operation, and with all inter-component relationships satisfied.

6 Future Work and Summary

In this paper, we described a method to customize a frontend appliance by making modifications to a standard system installer and coupling them with our graph-based system configuration infrastructure. Modifications to the sys-

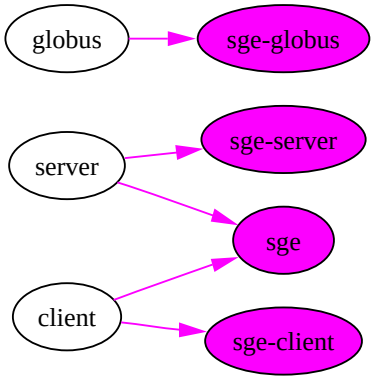

Figure 6: **SGE Roll Configuration Graph.**

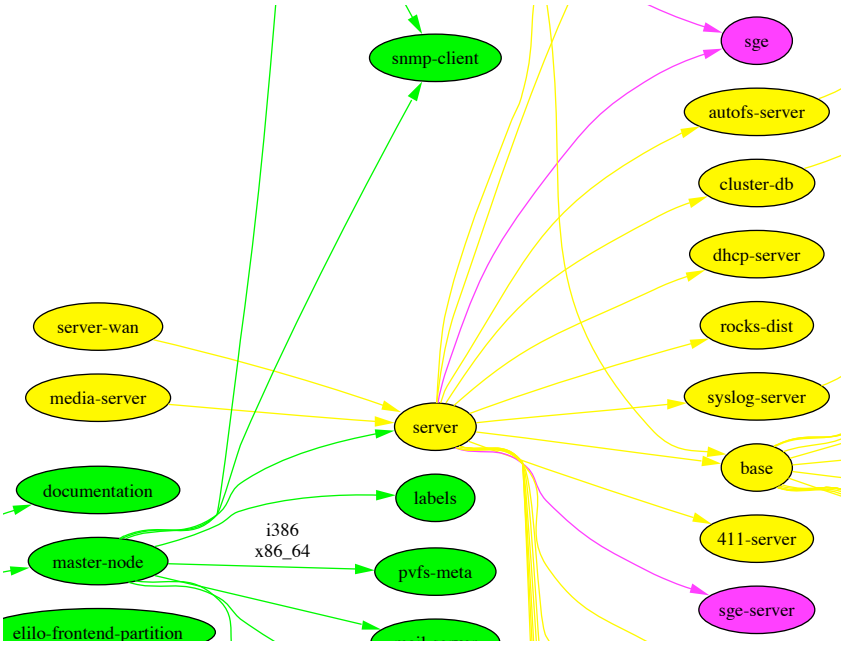

Figure 8: **Complete Core, HPC, and SGE Configuration Graph.**

tem installer were necessary in order to perform end-user customizations of the frontend during the initial build (as opposed to the *add-on* method which adds cluster-specific packages after a base OS is applied to a frontend machine). This modification enabled the addition and configuration of non-OS components in the controlled environment of the installer. With this method, we believe it guarantees consistent software configuration by reducing sources of uncertainty in the system. Currently, we only have anecdotal evidence to support this claim; formal methods need to be applied to our mechanisms to prove it.

Some have objected this method is too invasive: any incremental update to the software stack must be carried out with a complete reinstall. Users have also requested the ability to install a Roll separately from the OS, an add-on strategy. We are investigating methods in which to provide the controlled environment of the system installer on a running system. We are also in the process of characterizing what guarantees from our existing approach can be carried forward to an add-on/Roll hybrid strategy.

Although our method allows dynamic graph assembly, it relies on the Roll developer to specify connection points in both the core graph and possibly graphs of other Rolls. Currently we make heavy use of both the **server** and **client** vertices in the core graph as well-known graft points. However, by borrowing introspective techniques from languages like Java, we may be able to generate a richer list of well-known graft points and enable a more autonomous Roll development environment by minimizing the core graph knowledge required.

This paper also described how Rocks utilizes Rolls in order to export a cluster-aware Linux distribution. Other projects also use our method to provide services focused on specific communities. For example, the Geoscience Network [3] use Rolls to build a common environment in a grid of clusters to help researchers understand the complex dynamics of Earth systems. The Biomedical Informatics Research Network [8] similarly use the Roll mechanism to distribute software components that process brain imaging files in order to study human neurological disease and associated animal models.

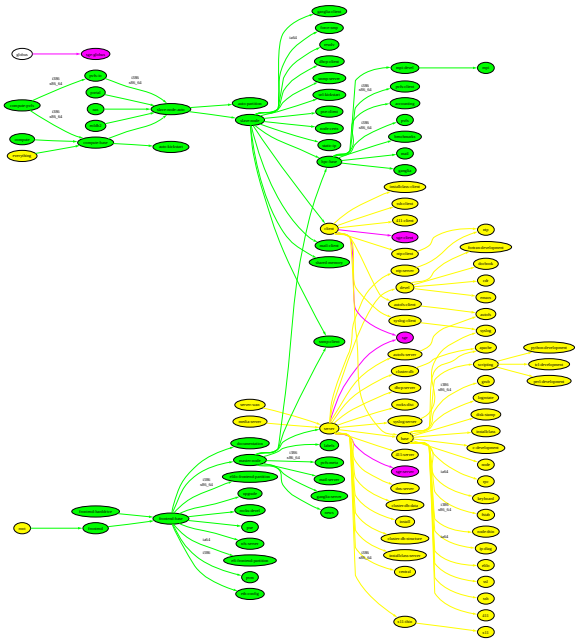

Figure 7: **Fully assembled configuration graph (Core, HPC Roll, SGE Roll).** This figure is purposely greeked to show the full graph with Figure 8 zooming in on the center of this figure. The Core is represented by the yellow vertices, the HPC Roll is green and the SGE Roll is purple.

More information about Rocks and downloadable ISO images for the Rocks Base and all the Rolls mentioned in the paper can be found at `http://www.rocksclusters.org`.

7 Acknowledgments

We gratefully acknowledge the support of Sun Microsystems who graciously donated a 128-node v60x cluster to support our cluster development.

We would also like to thank two of our Roll developers. Thank you to Roy Dragseth of the University of Tromsø for developing and maintaining the PBS Roll and to Thadpong Pongthawornkamol of Kasetsart University for developing the SCE Roll.

References

[1] Parallel and distributed systems software laboratory, rwcp. http://pdswww.rwcp.or.jp/.

[2] Scyld beowulf. http://www.scyld.com/.

[3] Karan Bhatia, Memon Ashraf, Ilya Zaslavsky, Dogan Seber, and Chaitan Baru. Creating grid services to enable data interaoperability: An example from the geon project. November 2003.

[4] Jack J. Dongarra. Performance of various computers using standard linear equations software (linpack benchmark report). Technical Report CS-89-85, University of Tennessee, 2004.

[5] Richard Ferri. The OSCAR revolution. *Linux Journal*, (98), June 2002.

[6] Erik A. Hendriks. BProc: The beowulf distributed process space. In *16th Annual ACM International Conference on Supercomputing*, June 2002.

[7] M. J. Katz, P. M. Papadopoulos, and G. Bruno. Leveraging standard core technologies to programmatically build linux cluster appliances. In *Proceedings of 2002 IEEE International Conference on Cluster Computing*, Chicago, Il, October 2002.

[8] Abel Lin, Patricia Maas, Steve Peltier, and Mark Ellisman. Harnessing the power of the Globus Toolkit. 2, January 2004.

[9] P. M. Papadopoulos, M. J. Katz, and G. Bruno. NPACI Rocks: Tools and techniques for easily deploying manageable linux clusters. In *Proceedings of 2001 IEEE International Conference on Cluster Computing*, Newport, CA, October 2001.

[10] Philip M. Papadopoulos, Mason J. Katz, and Greg Bruno. NPACI Rocks: tools and techniques for easily deploying manageable linux clusters. *Concurrency, Practice and Experience*, 15(7-8):707–725, 2003.

[11] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[12] Thomas L. Sterling, John Salmon, Donald J. Becker, Savarese, and Daniel F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, 1999.

[13] David A. Patterson Thomas E. Anderson, David E. Culler. A case for networks of workstations: NOW. *IEEE Micro*, February 1995.

[14] P. Uthayopas, T. Angsakul, and J. Maneesilp. System management framework and tools for beowulf cluster. In *Proceedings of HPCAsia2000*, Beijing, May 2000.

[15] The Warewulf Cluster Project. *http://warewulf-cluster.org/*.