BCFG: A Configuration Management Tool for Heterogeneous Environments

Narayan Desai
Andrew Lusk
Rick Bradshaw
Rémy Evard
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
{desai,alusk,bradshaw,evard}@mcs.anl.gov

Since clusters were first introduced[5], node counts have increased rapidly. Currently, a variety of clusters with more than one thousand nodes are listed on the TOP500 list. In the next three years, clusters with more than four thousand nodes are expected. Cluster management functionality has lagged behind all areas of system software. In order to effectively manage the clusters of today and tomorrow, the basic cluster management software model must change. Current techniques focus on the management of single nodes, as opposed to complete cluster configurations. This approach typically leads to automatic management of compute nodes, while using ad-hoc techniques to manage service nodes.

Configuration management is the process where software configurations on clients are installed, updated and verified. Scalability in this context applies not only to node count, but also to numbers of administrators and discrete configurations. Another important area is that of cluster-aware configuration description languages. We believe these concerns will impact large cluster operations greatly if not addressed.

To address these issues, we have begun development of BCFG, a symbolic configuration management tools for heterogeneous clusters. It uses a multi-tiered configuration description, allowing high levels of reuse among differing configurations. BCFG is intended as a vehicle for research into system management problems, and experimentation with new techniques. This effort was also motivated by our experiences on Chiba City[2], our testbed cluster. Testbed users can specify the software configuration for nodes; this allows the automatic replacement of any portion of the default cluster OS. Chiba City's configuration management requirements are complex, due to this testbed usage model. However, we feel that these problems translate to large scale computational clusters as well; as the size of these clusters grows, the numbers of discreet configurations employed will grow as well.

1. Motivation

Many configuration management tools are widely used. These tools fit into two basic categories: those that are metadata based, and those that are binary image based. Metadata based configuration tools use an abstract configuration description to create an on-disk configuration corresponding to the test description. Examples of this approach are Redhat's Kickstart[4] or Cfengine[1]. A simple text file configuration is used by the tool to provide automated installation. The end result is an on-disk configuration that corresponds to the specified configuration. In binary imaging systems, like SystemImager,[3] the server side configuration consists of a binary image. This image is not abstract, and contains the verbatim data to be installed on the client. The client's final on-disk configuration will be identical to the image on the server.

Metadata based systems use configurations that

can be decomposed into smaller usable pieces. In many cases, these fragments can be combined to form new configurations. This allows new configurations to be created quickly. Multi-administrator systems can assign modification privileges to configuration fragments so that administrator roles are enforced. Metadata based configurations are frequently portable between operating system releases and platforms. For example, simple Kickstart configurations can be moved between minor OS upgrades unchanged. Structurally, metadata configurations are usually formatted as a set of details about a client. This structure causes an important shortcoming; most metadata configurations are not comprehensive. Administrator interactions in these systems typically occur at an extremely abstract level. This can cause problems for administrators with specific problem solving styles.

Image based systems have a different feature set. Binary image configurations are comprehensive; all files included in any client are contained in the binary image. However, there is one caveat; because all clients contain host-specific data, binary imaging tools are insufficient to completely configure a client. Typically, a metadata-based configuration tool is used to complete the configuration process. With this addition, the formerly comprehensive binary image only describes a portion of the configuration. Due to the binary image format, configurations cannot be decomposed in any way. Because of the binary content of the image, data cannot be easily reused when changing platform or operating system version. Because the configuration description used by binary imaging systems is identical to a file system hierarchy, administrator interactions with imaging systems are similar to those with a single system. For this reason, administrators are easily able to start using these systems.

As shown above, both metadata and imaging based configuration management systems provide useful administrative functionality. The composability and reusability of metadata based systems dramatically reduce the creation time and size of different configurations. Imaging systems provide a comprehensive configuration description and an appealing user interaction model. As all of these properties are desirable, we have taken an approach with BCFG that attempts to blend useful features from both systems.

2. Approach

BCFG has been designed as a metadata based configuration system. While this approach exposes it to common metadata based system pitfalls, we have designed a configuration language that avoids many of these problems. Other important features, validation and change detection, and generators ease the integration process into existing administrative groups and environments. First we discuss the requirements, then the approach taken with BCFG.

2.1. Configuration Language

Metadata based configuration management systems use symbolic descriptions to construct a literal configuration. This construction process is at the heart of many problems with these systems. However, care has been taken when designing BCFG's configuration language to allow for extensive configuration composability, reusability, and verification. This goal impacts the configuration language in a number of ways. First, imperative operations must not be directly exposed to the user. This limitation is required because validation becomes difficult, if not impossible to achieve if users are allowed to specify arbitrary, opaque operations as a part of a client configuration. Second, a client configuration description must be comprehensive. By this, we mean that all files on the system must be accounted for in some way. Comprehensiveness is prerequisite for useful validation; if the entire system configuration isn't captured by the configuration management system, the validation domain will be severely limited. Consequently, the utility of validation is greatly reduced, as changes to unspecified regions of the system configuration cannot be detected. Third, the system must be extensible, as configuration patterns vary greatly from site to site. If a pattern isn't describable using the configuration language of a tool, then another tool will be written to handle the specific task. This will lead to multiple tools maintaining parts of system configuration, and ultimately to validation problems. Next, configurations need to be easily decomposable and recombinable. This ability allows configuration fragments to be flexibly recombined in order to create new configurations quickly. Finally, a binary configuration on a client needs to be invertible. It must be possible to generate the server side configura-

tion using the client's running configuration. The inversion process can be used to generate a symbolic description of detected changes in client configuration.

In satisfying these requirements, the configuration language used by BCFG took on some peculiar properties. Only client state is described. This means that client configurations consist of configuration fragments that capture the totality of client configuration. Configuration fragments are high-level abstract types common to all system configurations. Examples of these are packages, services and configuration files. Configuration fragments are grouped into bundles, groups of interdependent fragments that achieve a common task. Bundles can be arbitrarily combined to form new configurations. Client state correction logic is contained in the BCFG client executable. BCFG configuration descriptions are treated as comprehensive. Extensibility is implemented using the generator interface. (See 2.3 for details) The only configuration language requirement for generators is that automatic generation of configuration description can occur. Finally, inversion is enabled by BCFG's state description style configuration language. Configuration fragments can individually be verified and changes can be detected. (See Section 2.2 for details) If imperative actions were exposed to users, this would not be feasible.

2.2. Validation and Change Detection

The single weakest point of metadata based systems is the administrative model. Operations are complex, and in many cases abstracted away from desired results. Administrators spend time working on a configuration to fix a particular problem, and then will need to install their changes properly into the configuration management system. This can be a very error-prone process, as it is easy to miss necessary changes if the reconfiguration was complex. The addition of rigorous validation and change management and detection features would dramatically improves usability. This in turn enables a variety of new system interaction models.

Ideally, the configuration management system would be able to detect and incorporate changes made to a client configuration, as in imaging systems. It would then need to interrogate the user in order to properly incorporate this reconfiguration symbolically into the configuration description on the server. For example, if an administrator configured a client as a web server, the changes would need to be associated with a "web server" attribute.

The primary user interface to BCFG remains direct modification to the configuration description. However, automatic change detection functionality is also available. With the addition of a stringent verification process, changes can be reliably detected. Many metadata based system incorporate some type of configuration verification. This process usually consists of independent verification of all configuration aspects contained in the configuration description. The result of this check is to ensure that the client configuration contains the entire configuration specified by the configuration description. BCFG adds an additional step to this verification process. As the BCFG configuration language allows a client configuration to be inverted into configuration description, extra configuration fragments can be detected. This process is called two-way verification, as it first verifies that the client configuration contains no less than the configuration description specifies, and also verifies that the client configuration contains no more than specified. This capability allows BCFG to be effectively used in larger administrator groups, as all changes can be detected, even if systems have become configuration-skewed.

Change detection is enabled by a few system management features supported by most operating system distributions. Many systems use software packaging tools which store checksums of files contained in packages. This data can be used to detect reconfigurations of packages on the system. These packaging systems can export a list of currently installed packages. This list can be trivially compared to the configuration description to ensure that the correct list of packages is installed. Most systems have some manner of service startup control system. This system can be used to ensure that the proper services are started at boot time. Detecting changes in the aforementioned areas is an easy process. This leaves two important types of configuration fragments that are harder to automatically detect: configuration file changes and filesystem operations. Heuristics are used to detect changes in these two areas. Configuration file changes are the easier of the two to detect; partial detection is provided by package checksums. Files not owned by any package that are located in `/etc` are also detected. Similarly, filesystem changes, like directory

creation or permission changes, need to incorporate information from the packaging system database as well as manual searches of the client filesystem. `/etc` and `/var` are checked currently for new entries. While we may eventually scan other filesystem locations for configuration files as well, scanning a single location has proven sufficient for now.

The second part of automated change integration is the process of serializing changes and incorporating them into the configuration description. While changes can be detected, automatic incorporation is not yet functional.

2.3. Generators

Frequently, sources external to the configuration management system are canonical for certain types of data. This may be the case because of site design; a user management or host management system may already be in use. It may also be the case because of complex, dynamic configuration requirements. In either of these cases, the configuration management system must provide an interface for external access.

BCFG provides an interface in which generators can be implemented. A generator is program that can construct a configuration fragment based on task-specific logic. The generator API is a calling convention; generators are called during the client configuration generation process, and are given all available client metadata. As generators are treated as opaque executables, (that construct well-formed configuration fragments) they can access arbitrary data sources appropriately without configuration management system modification.

3. Results

BCFG has been deployed on a 320 node testbed cluster. Basic configuration description reusability has been verified by producing a redhat 9.0 image based largely on a redhat 7.3 image. Robust configuration verification has proven very useful. When developing a complete image for the first time, BCFG located a number of client reconfigurations that occurred as a result of automated installation processes. Formerly, these sort of changes would have remained undetected. The generator interface has also worked very well in our dynamic environment. Rapidly changing configurations can be easily modelled, and data from external systems can be easily integrated into client configurations. For example, generators have been used to maintain dynamic user access to nodes, in conjunction with divisional user account management facilities. Cryptographic key management facilities have also been implemented using the generator interface.

4. Future Work

A number of useful features remain to be developed. The most important of these is automated configuration change integration. Undoubtedly, our configuration change detection heuristics will need improvement. The addition of a change reporting interface will allow the calculation of clients affected by configuration changes. This enables sparse client reconfiguration when configuration changes occur.

Acknowledgements

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, SciDAC Program, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

References

[1] M. Burgess. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3), 1995.

[2] R. Evard, N. Desai, J. P. Navarro, and D. Nurmi. Clusters as large-scale development facilities. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER02)*, pages 54–66, 2002.

[3] B. E. Finley. VA SystemImager. In USENIX, editor, *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10–14, 2000, Atlanta, Georgia, USA*, pages ??–??, Berkeley, CA, USA, 2000. USENIX.

[4] Redhat linux customization guide, chapter 7, kickstart installations. http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/custom-guide/.

[5] 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 *International Conference on Parallel Processing, Vol.1: Architecture*, pages 11–14, Boca Raton, USA, Aug. 1995. CRC Press.