Directing Change Using Bcfg2

Narayan Desai, Rick Bradshaw, Joey Hagedorn, and Cory Lueninghoener
– Argonne National Laboratory

ABSTRACT

Configuration management tools have become quite adept at representing target configurations at a point in time. While a point-in-time model helps with system configuration tasks, it cannot represent the complete scope of configuration tasks needed to manage a complex environment over time. In this paper, we introduce a mechanism for representing changes over time in target configurations and show how it alleviates several common administrative problems. We discuss the motivating factors, design, and implementation of this approach in Bcfg2. We also describe how this approach can be applied to other tools.

Introduction

Change is unavoidable in today's complex computer networks. Changing user demands, security vulnerabilities, and external resource integration necessitate frequent reconfiguration of local machines. With the size of networks continuing to grow, there is no reason to expect the rate of configuration changes to decrease.

Three factors affect an administrator's ability to keep pace with the constant demand for configuration changes: efficient deployment of changes, control over how and when new changes are used and deployed by the server, and the ability to understand patterns in configuration changes and propagation. Each of these factors is needed in any tool that comprehensively supports change management.

First, the cost of creating and deploying configuration changes must be small, and the deployment of configuration changes need to be controlled. If each change takes several hours of focused administrator time, it cannot be deployed quickly. Similarly, the administrator must be able to control how configuration changes propagate to the environment from the configuration specification. To address this issue, we implemented and deployed Bcfg2 [3]. We presented an account of the deployment process and the resulting efficiency improvements at LISA last year [5]. From these experiences, we feel we have a reasonable solution for this aspect of the problem.

Once changes can be effectively described and deployed, administrators need to be able to represent changes over time. In frequently changing environments, administrators must be able to control when changes go into effect and how they are deployed. Administrators are currently forced to closely monitor configuration systems when performing complex reconfiguration workflows. This process is quite error prone and can cause serious system faults when it goes awry.

With configuration and system changes explicitly described, administrators have access to a wealth of information about the ebb and flow of the configuration. Through analysis of this data, administrators can characterize the underlying patterns in configuration changes and change propagation information. We believe this information will be of great use to administrators.

In this paper, we discuss our approach to simplifying change management procedures. In particular, we describe the modifications made to Bcfg2 and show how practical situations benefit from this approach. We also detail how other tools can implement similar solutions.

Background and Related Work

Bcfg2 is one of several configuration management tools that provide the user with a declarative interface to system configurations. Declarative tools allow the user to describe the goal configuration state, as opposed to a set of steps that will produce this result. They produce a set of reconfiguration operations that will result in the proper outcome. LCFG [2] was the first system to employ this model; more recently, Puppet [7] has also used it. Each of these tools uses a central specification to describe the desired configuration of a network of machines. Both tools have been used with repositories under version control; however, neither tool actually integrates with version control processes. We believe that revision control techniques could be leveraged for much more than just change logging and roll back.

System administrators have developed several techniques for dealing with change. Many of these approaches are manual. Many complex changes require manual orchestration and are very fault-prone.

Security audit tools have also become popular in the past several years. Tripwire [6] and Aide [1] are both popular auditing tools. These tools have a rigorous approach to detecting system configuration changes but employ a filesystem-based approach to detecting system changes. While this approach helps administrators locate misconfigurations, it does not integrate with an overall model of system configuration.

Some researchers have studied the costs of system administration. In [4], the authors present a cost model of system administration. They found the application of real quantitative data resulted in reinforcement of several intuitive results and found several interesting new patterns. We feel the availability of more quantitative metrics about the configuration process would improve administrators' decision-making and problem-solving abilities.

Approach

The goal of this work is to augment Bcfg2 in order to explicitly represent changes in configuration specification and client configurations over time. Without a notion of time in the system, administrators can interact only with the current state of clients. They can neither analyze past events nor orchestrate future changes. Both of these capabilities are needed if reliable fault recovery mechanisms and scalable administration processes are to be implemented.

As these detailed statistics are collected, a large pool of data accumulates that can be used to better understand long-term trend information and change propagation to clients. This result is a model for the entire life-cycle of systems, including their point-in-time configurations, reconfigurations, and misconfigurations.

In this section, we discuss our approach in detail. We begin by detailing the motivation for this work. Next, to lay a foundation for our solution, we present a high-level overview of the Bcfg2 architecture. We then discuss our implementation in Bcfg2.

Motivations

Configuration management systems seek to efficiently represent a goal for a large group of somewhat similar client systems. Once we felt that we had a good implementation of such a system, it was only natural to extend the representation into the past and future. We were motivated by several operational difficulties. These fell into three main categories:

- Performing reconfiguration workflows of interdependent clients
- Enforcing change management policies
- Providing comprehensive audits of past client configuration states

While tasks of these sorts could be represented, pointwise, by using Bcfg2, their implementations were time-intensive and highly fault-prone. We felt that each of these tasks could be greatly simplified with explicit support from Bcfg2.

Bcfg2 Architecture

Bcfg2 is structured as a client/server application with three main parts: the server, the client, and a reporting system.

The Bcfg2 server houses a configuration specification that describes all aspects of configuration for all managed clients. It uses this specification to build per-client configurations when they are requested. It also provides all network services required by the Bcfg2 client. The configuration specification is stored in a filesystem hierarchy. The server daemon uses FAM to monitor file system changes so as to efficiently cache specification date in memory. The resulting system has low overhead, as it only interacts with the filesystem when modifications require. The server also produces a record of statistics describing clients, including their current states and Bcfg2-related activities.

The Bcfg2 client consumes the client-specific target configuration and performs all operations on client systems. It analyzes the current state of the client system, compares that state with the target configuration, and produces a set of operations that must be performed in order to reconfigure the client system into the target state. Once the Bcfg2 client has performed these tasks, it uploads a set of statistics describing the results of its operations to the server.

The Bcfg2 reporting system postprocesses the client statistics collected by the Bcfg2 server. It produces textual reports, delivered as emails, Web pages, or RSS feeds. These reports are used to display current client conformance with the configuration specification. Administrators can use these reports to repair systems with incorrect configurations or include new client configuration aspects in the specification. While the ability to describe and propagate a desired configuration is independently useful, the reporting system has proven to be the most critical feature offered by Bcfg2. It allows fluid reconciliation of reality with administrative goals, since reality rarely plays along.

Implementation

Addressing these issues required modifications to all three parts of Bcfg2. We discuss each of these in turn, describing how the added functionality improves administrator control and understanding, and interacts with the other parts of the system. In particular, we realized that tracking time stamp information at these three critical points would provide us with the range of functionality we needed. These modifications provided the infrastructure to implement our solutions discussed in the next section.

Bcfg2 Server

We modified the Bcfg2 server to integrate with a configuration repository managed by Subversion [8]. This integration enables the Bcfg2 server to query the repository for the current subversion revision. The current repository revision is included with all client configurations generated by the server. Upon each update to the repository, the revision is updated. We also added a revision log to the server. This log tracks the repository revision used by the server at all times and can be used to determine which revision of the repository was in use at any past time.

This approach has three main benefits. First, administrators can use Subversion to manage their

configuration specification. The benefits of version control are well known and will not be discussed here. Second, the repository revision number provides a discrete set of configuration timesteps that are explicitly tied to repository contents. Third, administrator intent is documented by the revision log. This allows one to determine the desired configuration state of a client at any earlier time.

In practice, changes are made to a separate checkout of the repository and committed to the master repository. The server can then run any revision of the repository, regardless of the current contents of the HEAD branch. The server repository revision is controlled by a discrete utility. This can be used to upgrade or downgrade the server's copy of the repository.

Bcfg2 Client

We modified the Bcfg2 client to process revisions included in configuration specifications. Each revision is associated with all client statistics and uploaded to the server. Because a client can reconfigure at any time, having a change-based discrete time step as a revision number is essential. This approach avoids the need for the retention of any client-side state.

Bcfg2 Reporting Subsystem

We modified the reporting subsystem to retain all configuration statistic records. Previously, it retained only the newest record and the last record in which the client was correctly configured. We also enhanced it to store all statistics with revision information. We currently have a series of reports that summarize this information in basic ways. Over time, we will develop other reports as we find useful views of the data. Data from the reporting system is the basis for all configuration feedback in Bcfg2.

Results

Overall, we have been pleased with the results of this approach. It has exposed a variety of quantitative metrics that we use to better understand the configuration process on our systems. We have used this information and the enhanced control facilities to implement solutions to a number of subtle administrative problems. In this section, we discuss how these facilities lead to dramatic improvements for three broad classes of problems: change orchestration, or the coordination of changes across several systems to achieve a uniform goal; software enforcement of change management procedures; and analysis of past configuration states and changes. Each of these areas was poorly served by previous generations of configuration management tools. For each, we define the problem and provide a concrete example. We describe how the change-based features of Bcfg2 enable the implementation of reliable solutions to each of these issues. Each of these issues can be very time-intensive in large administrative groups or complex environments, so the potential benefit of each is substantial.

Solutions in each of these areas have become part of the daily administrative process. Administrators have found these techniques quite useful and are using them to reach new heights of productivity. Several frustrating tasks have now been automated, so administrators are more contented as well.

Change Orchestration

Frequently, configuration goals require the reconfiguration of several systems in a coherent fashion. These operations are tightly coupled; that is, operations must be performed in the proper sequence and are contingent on the successful completion of other operations. Misordered operations can result in a number of bad outcomes, ranging from transient failures to overall system failures. These workflows are needed in several common situations. When services are changed, clients must be reconfigured to use new services. Moreover, these services may only be used once they have been properly configured. Likewise, all clients must be removed from services prior to their decommission.

In order to automate this process, we have used repository revision numbers to represent states in a finite state machine. Administrators begin by describing all discrete states as individual repository revisions. Administrators then construct an explicit state diagram detailing how the workflow can proceed. Each workflow step consists of two parts. First, the server begins using a particular version of the repository. Then clients are reconfigured using this version. If all clients configure properly, then the system can proceed to the next step in the workflow. If they fail, the server proceeds to the failure result state for this operation. We have written a script that implements this process using data from the reporting system to determine when clients have successfully entered a state. This process can be composed to produce complex series of operations, complete with failure contingencies.

The technique has two important limitations, however. While operations can be chained, administrators must map out all contingencies into discrete states. This process becomes complicated and time consuming in the face of large combinations of failing and succeeding operations. Also, the time spent in any given state is not bounded. That is, success or failure of an operation may be contingent on the actions of a client that is down or not operating properly. To mitigate this issue, administrators can limit per-operation checks to a series of machines that are pertinent to the operation. While this approach is not universally possible, it makes state checks for many operations much simpler. Also, administrators can query a workflow for blocking issues. This technique allows the to locate clients in need of manual intervention.

Change Management

Change management is a set of techniques that ensures changes are performed in a systematic fashion. These policies are useful throughout the change

process, from the initial creation of changes through the testing and activation of these changes. Change management is essential to guarantee the quality and reliability of changes made on production systems. In short, change management controls the conditions in which changes can be legitimately performed.

Change management policies tend to be site-specific. Many of the factors driving these policies are influenced by the reliability guarantees made to users and the ways that particular systems are used. We described initial work implementing change management policies in [5]; however, these processes were largely manual. Our enhancements to Bcfg2 have allowed us to automate most of the processes by implementing automatic enforcement mechanisms for these policies.

In our environment, two configuration management policies are in place. On our core infrastructure, changes are never performed automatically on critical servers. On one of our production clusters, major changes can be made only during a maintenance window, and changes can never be made while user jobs are running. For each of these policies, we describe the software implementations, our practical experiences, and the pitfalls involved.

Manual Change Deployment on Servers

In some cases, there are critical resources that should never be automatically reconfigured. Each of these machines is configured to run the Bcfg2 client in dry-run mode, through Bcfg2 itself. This policy is not hard to implement; however, it is hard to integrate into administrative processes. In many cases, critical systems will not receive configuration updates in a timely fashion because of the increased cost compared with other systems.

Bcfg2's reporting system provides a useful solution to this problem. All clients, even those running in dry-run mode, upload a set of current state statistics. These statistics can tell whether a machine is configured properly, has been configured recently, and has been configured against the most recent version of the configuration specification. Such information may be used to locate hosts that are either out of date or misconfigured. We use this method, in the form of a nagging email, to remind administrators about their critical machines. This approach has worked fairly effectively; administrators are given all of the information they need to address configuration problems on critical systems quickly.

Maintenance Window

Maintenance windows are a planned period in which changes can be made to systems. They are frequently used on production resources. The use of a weekly maintenance window is a requirement on one of our production clusters. This is required because of system policies and the need to maintain software consistency across all nodes that may participate in a single user job.

Our previous implementation of this policy consisted of a choice between two suboptimal solutions. Without fine-grained repository control, administrators were forced to choose between prepopulating the repository with changes and making the changes synchronously during the maintenance window. Both options have serious drawbacks. If changes are made to the repository prior to the maintenance window, they can be consumed prior to the window. That is, clients that run Bcfg2 in dry-run mode will verify against the wrong configuration and will be marked incorrect. This spurious failure can mask legitimate configuration problems. Also, clients that are rebuilt during this interval will be misconfigured, making the configuration inconsistent across the cluster. If changes are made synchronously, administrators are forced to perform all specification updates during the maintenance window. This can be time-consuming and easily result in forgotten changes that cannot be deployed until the next window. Our previous solution was to run the Bcfg2 client in dry-run mode, except for during the maintenance window, when changes could be made. While this produced the correct result, it was unwieldy for administrators.

Correctly addressing this problem requires two capabilities. First, administrators must be able to conveniently queue changes for later use. Once the repository is stored under version control, this operation becomes trivial. Second, the repository must not be changed between maintenance windows. We modified the repository control script described in the Results section to accept a configuration file describing maintenance window times. This script will allow repository changes only during the specified windows. It can be overridden, if circumstances warrant; however, the script prevents simple mistakes from violating change management policies.

Change Analysis

Experienced system administrators have an intuitive sense for how often configuration changes are performed, whether they result in client changes, and whether those client changes have been deployed. While these are useful instincts, they can be based on flawed or incorrect information. Quantitative metrics would provide a more solid foundation for decision making and problem solving.

We have implemented several reports that summarize change information, on both the client and server sides. From these, administrators can better understand rates of specification change, client change, and change propagation and can understand actual patterns present in their systems. We believe the availability of this information will result in sounder change management policies.

We have also implemented reports that construct timelines of configuration operations. These timelines

aid in problem solving. Users often report new failures after a large number of configuration changes. Change summaries are now readily available, enabling administrators to find likely sources of problems. Alternatively, if a system compromise is discovered, configuration logs can be used to determine whether other systems were susceptible at the time of the initial attack. Moreover, this information can be used to determine how long hosts were vulnerable and when they were initially patched.

Conclusions

We believe this approach to be a vital step toward a long-term goal of complete integration of configuration management into the administrative process. Change occurs frequently in most environments; its omission from the configuration management model is a serious oversight that inhibits the development of system configuration tools in a number of beneficial directions.

Status and Future Work

The Bcfg2 codebase is publicly available and is released under the GPL. It is used at a number of sites worldwide. Source code, documentation, papers, presentations, and mailing list information are available at the Bcfg2 Web site [3]. The features discussed in this paper are in use at Argonne currently and should be included in a stable release by early summer. The information exposed by this work is broadly applicable to the analysis of a number of complex issues in system administration. For that reason, it is impossible to predict which uses will bear fruit. We can, however, suggest several interesting possibilities. Static analysis of history information could reveal a number of interesting patterns in system configuration histories. For example, it could locate clients that are frequently misconfigured or remain misconfigured for long periods of time. Similarly, the deployment of critical security updates can be tracked, producing a list of known insecure hosts. All of these factors can be used to produce sophisticated risk assessments.

Similarly, fine-grained control over change deployment provides a powerful infrastructure for tools to build on. Developments in this area will also be guided by site-based needs. This capability is a clear prerequisite for reliable autonomics. We plan to integrate some basic diagnostic functionality into our current generation of scripts to experiment with autonomic principles. In a similar vein, this work allows the implementation of network unit tests. In principle, it could be used to implement deployment regression tests with automatic backout in the case of failures.

Also, our current implementation of configuration workflows have several serious limitations. We plan to experiment with other models to see whether these limitations can be mitigated or eliminated altogether.

Change Support in Other Tools

We believe this approach to be applicable to any declarative configuration management tool. These tools require two modifications in order to implement the full range of functionality described in this paper. However, the first can be implemented without the second. First, the tool must tightly integrate with a repository under revision control. Revision tracking by the server provides the basis for a historical view of the configuration specification. This provides many of the change management benefits described above.

The second step is to include revision information in any statistics collected by the system. This feedback allows the configuration management tool to detect the entry of clients into particular states. Integration with a feedback system is vital to support the change orchestration functionality described above. In the long term, this functionality will be required by autonomics facilities as well.

Author Biographies

Narayan Desai has worked for several years as a systems administrator and developer in the Mathematics and Computer Science Division of Argonne National Laboratory. His primary focus is system software issues, particularly pertaining to system management and parallel systems. He can be reached at desai@mcs.anl.gov .

Rick Bradshaw holds a BS in Computer Science from Edinboro University. He has been a memeber of the MCS Systems team since 2001, where he aids in maintaining HPC resources, experimental computing resources, and general UNIX infrastructure. He can be reached at bradshaw@mcs.anl.gov .

Joey Hagedorn is a student at the University of Illinois, Champaign-Urbana. When not studying, he works on several software and hardware projects. He can be reached at hagedorn@mcs.anl.gov .

Cory Lueninghoener earned his MS in Computer Science from the University of Nebraska-Lincoln, where he worked with the Research Computing Facility. He now works with the HPC Administrator team at Argonne National Laboratory, currently focusing on Argonne's Teragrid cluster. Cory can be reached at lueningh@mcs.anl.gov .

Software Availability

Bcfg2 is an open source project mainly developed at Argonne National Laboratory. Source code, documentation, binary packages, mailing list archives and more are available from the Bcfg2 web site [3].

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific

Computing Research, U. S. Dept. of Energy, under Contract W-31-109-Eng-38.

Bibliography

[1] *Aide Web site*, http://www.sourceforge.net/projects/aide/ .

[2] Anderson, Paul and Alastair Scobie, ‘‘Large scale Linux configuration with LCFG,’’ *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, pp. 363-372, October 10-14, 2000.

[3] *Bcfg2 Web site*, http://trac.mcs.anl.gov/projects/bcfg2/ .

[4] Couch, Alva L., Ning Wu, and Hengky Susanto, ‘‘Toward a cost model for system administration,’’ *Proceedings of the Nineteenth System Administration Conference (LISA XIX)*, San Diego, CA, December 4-9, 2005.

[5] Desai, Narayan, Rick Bradshaw, Scott Matott, Sandra Bittner, Susan Coghlan, Reémy Evard, Cory Lueninghoener, Ti Leggett, J. P. Navarro, Gene Rackow, Craig Stacey, and Tisha Stacey, ‘‘A case study in configuration management tool deployment,’’ *Proceedings of the Nineteenth System Administration Conference (LISA XIX)*, San Diego, CA, December 4-9, 2005.

[6] Kim, Gene H., and Eugene H. Spafford, ‘‘The design and implementation of tripwire: A file system integrity checker,’’ *ACM Conference on Computer and Communications Security*, pp. 18-29, 1994.

[7] *Puppet Web site*, http://reductivelabs.com/projects/puppet .

[8] *Subversion Web site*, http://subversion.tigris.org/ .