Fully Distributed Service Configuration Management

Paul Murray, Patrick Goldsack
Hewlett-Packard Laboratories
pmurray@hp.com, patrick.goldsack@hp.com

Abstract

Configuration management in today's data centers is largely a human activity. Where automation does exist it is usually implemented by centralized management tools that coordinate configuration actions across the entire infrastructure and applications. These systems are limited in scale, reliability, and security.

We propose that dependable service configuration management is more naturally implemented by a loose federation of fully distributed and self discovering management systems that interact through controlled information exchange.

1. Introduction

Service configuration management in a data center environment is an ongoing activity involving deployment, modification and ultimately removal of distributed software systems and underlying infrastructure.

Centralized configuration management uses a central control process to manage the configuration of a distributed system remotely. Coordination of actions across the configured elements is simple as it is handled from within a single process. However there are significant problems. The centralized configuration function is an additional element that introduces a point of failure. It also needs to be configured and managed. It interacts with all system elements and so has widespread security risks if compromised. And its scale is limited to that which the central process can handle. The centralized approach is well understood and easy to implement, but as the scale of systems increases it becomes ever more inappropriate.

We believe the other extreme, fully distributed configuration management has more to offer our understanding of how to build future configuration management systems.

In fully distributed configuration management each element has its own configuration capability. Coordination of configuration across the elements is more complicated as it is a distributed act. However, there are advantages. The system is more resilient to failures. There is no separate configuration element to manage. Unrelated elements need not interact, so stronger isolation can be used to reduce security risks. And the total system scale is potentially unlimited.

Our experiments to date suggest that for reasons of dependability the fully distributed approach leads naturally to federated, self discovering management environments that interact through loosely coupled information exchange: firstly because self discovery leads to robust systems that handle change by design, and secondly because isolated management systems with limited interaction are simpler and offer improved security.

Here we concentrate on the dependability aspects that lead us to this conclusion, and discuss the implications for configuration management system design.

2. Configuration Management

We have developed a collection of tools that form the basis of our configuration management experiments. At the core of these tools is the SmartFrog framework. This framework is extended for dependability with the WoodFrog persistence mechanism and the Anubis state monitoring service.

2.1. SmartFrog

SmartFrog (described in [2]) is a software framework for describing and managing distributed software systems as collections of cooperating components. SmartFrog components are Java objects that are implemented by extension of SmartFrog core framework objects. These components can be configured and composed together using the SmartFrog description language. The core framework uses agents on each server to deploy systems by interpreting the descriptions, placing and constructing the described components, and transitioning them through their runtime lifecycles. At runtime the core framework provides a fully distributed naming and name-based reference resolution framework through which components can inspect and manipulate their configuration and interact with their environment. Components are also able to deploy further descriptions or terminate deployed components programmatically.

SmartFrog has been used for some years as a model based approach to distributed system configuration management. Examples are documented in [1], [3] and [4]. We refer to a deployed system as a live model as the framework reflects the system's current configuration as it undergoes changes throughout its runtime existence.

In [1], Anderson et al. demonstrated integration of SmartFrog with LCFG, a policy based system for installing and configuring UNIX servers, which has since been repeated with other OS neutral tools. The integration allowed LCFG policies to be managed as SmartFrog components, allowing software and server configurations to be deployed as components. With LCFG able to install the SmartFrog agents themselves, this provided the means for SmartFrog to not only deploy the application software described in a model but also as much management infrastructure as it needed.

Given further integration with other tools to configure virtual machines, networking and storage, one does not need to extrapolate far to see that the entire virtual infrastructure, OS, management systems and applications can be deployed on demand in the same manner.

The weakness of live models is that the loss of these runtime objects, due to failures, or even just rebooting a server, disrupts the operation of the framework. The WoodFrog and Anubis components address these issues.

2.2. WoodFrog and Anubis

WoodFrog (described in [9]) is an extension to the SmartFrog core framework that implements recovery from stable storage. It does this by maintaining a copy of the component's live model, representing its current configuration, on stable storage, and provides a means to re-deploy the component from this model.

The WoodFrog extension transparently rebinds references held by other components and extends the runtime component lifecycle to accommodate the offline status of a component that undergoes an outage.

Components defined by extension of the WoodFrog recoverable component can be included in any SmartFrog system model. This provides a means for the live models to overcome the limitation of only existing as runtime Java objects.

Anubis (described in [7]) provides the ability for distributed components to discover each other, monitor each others' states, and perform distributed failure detection. It is implemented as a distributed service provided by components that form a peer group and use group communication protocols to disseminate state information.

Anubis can be combined with WoodFrog to support the rebind feature by proactively disseminating updated references, and to provide more accurate failure detection to inform decisions about when to recover components.

A component can also use Anubis to indicate that it is about to undergo a deliberate outage, perhaps to reboot its server, by changing its status.

3. Architectural Principles

We have constructed several experimental systems using the fully distributed configuration management framework described above. Each was able to handle failures of the application, server, and networking by detecting the failure and automatically reconfiguring the service to compensate. Examples include the print service demonstrator reported in [1], the HP Utility Rendering Service [3], and the SoftUDC virtual infrastructure management prototype reported in [4].

Over the evolution of these prototypes we have recognized two principles that have simplified our systems and made them remarkably robust: self discovery and loose federation.

In the following we describe resource management in the HP Utility Rendering Service and how it exemplified these principles.

3.1. The HP Utility Rendering Service

The HP Utility Rendering Service [3] ran 12 separate instances of a CGI film rendering service for 12 independent film makers in a shared data center environment. The services all competed for access to a shared resource pool of 120 servers, allocated dynamically according to market based principles.

The resource allocation service and the 12 rendering services all operated as independent, distributed and self discovering systems. From the perspective of this paper, the primary interest is the way these separate services interacted to deal with dynamic configuration changes and failures using the resource management pattern described in [8].

Each computer in the system contained its own resource allocation component, forming a distributed resource allocation service. A single market based arbitrator was deployed into the resource pool.

An instance of the rendering service was deployed into the resource pool for each customer (film maker). This included a service manager that would control rendering jobs: sets of independent batch tasks (frames to render) that the manager could assign to a compute node.

Anubis was used as a general discovery and state observation service throughout. All compute nodes were configured to join the Anubis service and expose information about their allocation status. The resource arbitrator observed resources through it. The rendering service managers observed their resources and software components through it as well. Anubis was also used as the information exchange medium between the different configuration systems.

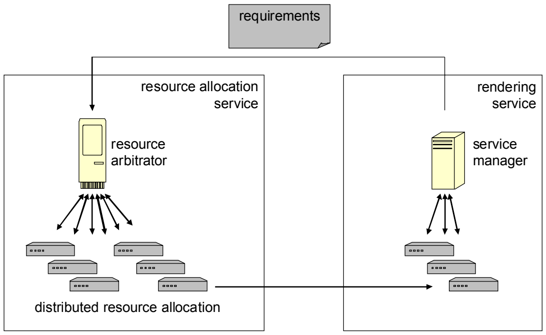

Figure 1: Service Interaction

Figure 1 shows the interaction both between and within the resource allocation service and one of the rendering services.

Each compute node was either free or allocated to a specific rendering service. The resource arbitrator would observe the resources and the requirements and use a market based scheme to propose matches. The compute nodes would observe these proposals, but were free to determine their own assignment.

A compute node would expose itself in a state space owned by a particular service instance. The service manager would discover the set of compute nodes allocated to it and, on finding a new one, would deploy code and data to that node (using SmartFrog) to provision it for the service.

The recovery models for the different parts of the system differed according to the role. Each service manager persisted information into stable storage using WoodFrog so that it could be recovered as required. The resource arbitrator worked purely from observed states and could be treated as stateless. The dynamic compute nodes hosting rendering engines lost their work in progress (the rendering software didn't checkpoint part-completed frames), so the recovery model was simply to allocate a spare node and to restart the computation.

Failure would be noticed by affected systems through their respective discovery mechanisms (the state space) ultimately leading to correction of the resulting incorrect configuration.

3.2. Self Discovery

The HP Utility Rendering Service provides a good demonstration of the dependability aspects of our framework. Each service was a self organizing collection of distributed components that discovered each other and their environment and responded to changes in an attempt to maintain their ideal configuration.

The underlying philosophy was to build a configuration management system for the utility that was responsible for discovering and reacting to its own environment, including discovering the availability of its own resources. Allocation of the resources was handled as a separate concern and was not directly exposed to the other services other than the fact that each service itself discovered its resources.

The resulting dependability of the HP Utility Rendering Service surpassed our expectations. During a 10 month period it ran 24 hours a day, encountering 154 failures due to rendering software and NFS and approximately 40 reboot failures (hang on reboot). All these failures were detected and handled automatically. Machines were permanently excluded on 19 occasions due to faults requiring human intervention to correct.

3.3. Loose Federation

There are many reasons for isolating subsystems, including security and division of responsibility, and we have repeatedly found that we have been able to separate configuration concerns and implement them as isolated services that interact through information exchange in a loosely coupled way.

If we separate out configuration systems in this way we find there is no natural single point of authority or control and any attempt to introduce one is artificial and unnecessary.

The HP Utility Rendering Service is an example of loose federation. The rendering services, were entirely self configuring, were not aware of each others existence, and did not interact. Similarly, the resource allocation service did not interact directly with the rendering services and was not known to them, or vice versa. The resource manager was only interested in resource requirements and resources. The rendering services were only interested in the resources allocated to them. None had authority over the others or played a subservient role. All interaction was indirect through the Anubis service.

The fact that Anubis pervaded the entire system somewhat contradicts our claims of isolation. In other examples we have provided much stricter isolation: in the SoftUDC for example, infrastructure resources allocated to separate service instances had no network connectivity between them. Interaction with the resource allocation service could only occur through a dedicated bastion host per service, and resources were only discovered when they were introduced to the same network by a network configuration service.

4. Related Work

Aspects of our approach are similar to autonomic computing as outlined in [5] and described architecturally in [10]. Although all the same principles are present, such as self-configuration, self-adaptation, and self-protection, we are less strict about the interaction among configuration systems. Rather than establishing contractual, service provider style relationships, we propose a much looser connection based on information exchange.

In their review paper [6] McKinley et al. identify assurance, security, interoperability and decision making as key challenges to adaptive software. We address these topics for configuration systems, describing our view of how they should be structured to separate concerns, localize decision making and control interaction.

Our claim of potentially unlimited scalability depends of course on implementation, but is upheld by the wealth of self-organizing peer-to-peer systems that have arisen. They are instances of fully distributed configuration management and both SmartFrog and Anubis use the same underlying principles.

5. Future Work

We suggest that loosely federated management systems based on this form of interaction are an ideal basis for data centre configuration management precisely because they support isolation and the lack of central control.

We believe this approach provides a robust and secure means to construct dynamic configuration management systems and we are currently working to extend these concepts to include all aspects of virtualized infrastructure and adaptive software systems.

It is essential to provide the means for these systems to interact in a secure way. The ongoing research in our group examines suitable interaction mechanisms to construct federated management systems for shared data centre environments, including the separation of responsibility, the security to prevent unauthorized configuration effects and maintain isolation, and the reliability to continue control of the system in the face of failures.

6. References

[1] P. Anderson, P. Goldsack, J. Paterson, “SmartFrog meets LCFG - Autonomous Reconfiguration with Central Policy Control”, *Proc. of the 2003 Large Installations Systems Administration (LISA) Conf.*, Oct. 2003

[2] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray and P. Toft, “SmartFrog: Configuration and Automatic Ignition of Distributed Applications,” *10th OpenView University Association Workshop*, June 2003.

[3] Hewlett-Packard, *Servicing the Animation Industry: HP’s Utility Rendering service Provides On-Demand Computing Resources*, http://www.hpl.hp.com/SE3D, 2004

[4] M. Kallahalla, M. Uysal, R. Swaminathan, D. Lowell, M. Wray, T.Christian, N. Edwards, C. Dalton, F. Gittler. “SoftUDC: A Software Based Data Center for Utility Computing”, *IEEE Computer*, pp. 46-54, November, 2004.

[5] J. Kephart, D. Chess, “The Vision of Autonomic Computing”, *IEEE Computer*, vol. 36, No. 1, 2003, pp.41-50

[6] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, B. H. C. Cheng, “Composing Adaptive Software”, *IEEE Computer*, Vol.37 No.7, July 2004, pp.56-64

[7] P. Murray, “The Anubis Service”, *Hewlett-Packard Laboratories Technical Report*, HPL-2005-72, 2005

[8] P. Murray, “A Distributed State Monitoring Service for Adaptive Application Management”, *Int. Conf. on Dependable Systems and Networks (DSN-05)*, June 2005, Yokohama, Japan, pp.200-205

[9] S. Rodrigo, P. Murray, “WoodFrog: A Persistence Library for SmartFrog Components”, *Hewlett-Packard Laboratories Technical Report*, HPL-2006-37, 2006

[10] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, J. O. Kephart, “An Architectural Approach to Autonomic Computing”, *Proc. of the Int. Conf. on Autonomic Computing*, May 2004, pp.2-9