

Adaptive provisioning using virtual machines and autonomous role-based management

Kyrre Begnum and Mark Burgess

Faculty of Engineering, Oslo University College, Norway

John Sechrest

EECS, Oregon State University, USA

Abstract

A modern server system must deliver a complex set of obligations towards its users. We study the concept of roles as autonomous encapsulations of a set of promises. Using virtualization and configuration management, every defined role for a real-life computer system can be realized as a self managing agent that is aware of other, related roles within the same broader context. A prototype implementation that provides web hotel services for customers is presented and discussed.

1 Introduction

In a network, computers are often assigned abstract roles in order to delegate tasks or functions within a collective organizational framework. This is reflected in typical descriptions like: *webserver*, *desktop*, *database* and *cluster node*. But what do we mean in practice by such a role? Where, in fact, are the true boundaries of a 'role' in network terms and the system that implements it?

In this paper, we use both theory and implementation to show that roles and systems have a natural definition that goes beyond single computers or particular services. We begin by discussing a model of interacting agents using Promise Theory[1] and go on to implement an adaptive scenario based on collaborating autonomous components.

Promise Theory is a model for describing complete networks of decision-autonomous components and imbuing them with relationships that allow them to work together. By working together, they can form a larger virtual entity, through the implementation of their policy promises.

Using this model, we show how the concept of a virtual system can be defined, independently of hardware or software, by looking at the natural regions of influence of collaborating promises. We compare this idea to a system running virtualized[6] GNU/Linux kernels that are managed by individual policy agents (cfengine), and finally show how theory directs us to build a survivable computing platform

using these commodity components.

2 Roles and boundaries

A network allows for organizational collaborations between computing components. Initially, computing devices belonging to multiple organizations or owners are *autonomous*, i.e. independent of decisions made, or services provided by other devices in the network. In order to work together, they must undergo a collaboration based on a principle of voluntary cooperation.

A natural reason for collaboration in networks is business. There are economic reasons for wanting to simplify the organization of networks by delegating responsibilities. There are also reasons to be able to purchase services on demand in times of additional need. This leads us to a view of systems based on network services.

A role is commonly associated with the provision of a service, as we mentioned above, however, we would like to show that this is only one kind of role. Roles are a generic way of creating a higher level abstraction for the configuration of system components. Services are only one aspect of this. A single computing device may take on several different roles in a network, depending on the services it provides.

A role is more than a service—it is a functional aspect of a network: it might imply a collection of services and configuration changes needed to make it function. To implement a set of behaviors, it may be necessary to describe related changes between different machines that have different roles. For example, if some part of the network is performing the role of “webserver”, it may require the gateway machine for the virtual network establish a hole in the the firewall, or establish a port forwarding rule to enable access to that web server. How can we use a model to capture these issues?

3 Promises and Service Level Agreements

Promise Theory is a representation of policy, from the viewpoint of completely autonomous components—i.e. agents with the power to decide their own behavior. No one can force a network node to perform any service or play any role in Promise Theory; if such a relationship is to emerge it must do so voluntarily, perhaps through an explicit economic agreement, like a Service Level Agreement. This is clearly an appropriate way to model a business environment where delegation (out-sourcing) and collaboration is desirable.

The strength of Promise Theory lies then, amongst other things, in its philosophy of making all assumptions about an agent's behavior into explicit and autonomous promise statements, and in tying a service view of interactions to a well-understood economic model.

Let us define the policy of an agent.

Definition 1 (Agent policy) *The policy of an individual computing device (agent or component) is a specification of its desired configuration, in terms of appropriate constraints on its behavior. Here, we shall use the example of a cfengine specification as policy.*

Promise Theory takes a service viewpoint of policy and uses a graphical language to compose system properties and analyze them. Its manifesto is to begin with detailed, *typed* promises, from which an algebra of cooperation can be understood, and then gradually to eliminate the specific types through transformations which bring out cooperative features of the network, such as reliability and economic transactions. The form of a promise is:

$$n_1 \xrightarrow{\pi} n_2 \quad (1)$$

where n_1 is the node making a promise, n_2 is the recipient of the promise and π is the typed promise body, which describes the interpretation and limitation of the promise.

A promise body is a constraint on behavior, asserted from one agent to another. It has a type (what is being promised) and a constraint (what subset of possibilities is being offered). For example, suppose the promise type from a server to a client is to provide WWW service, the promise body could represent a Service Level Agreement (SLA) to deliver data on demand within 5 milliseconds. Such promises are not uncommon. Promises therefore offer us a language of 'primitives' for composing service policies.

Promises fall into a few notable categories:

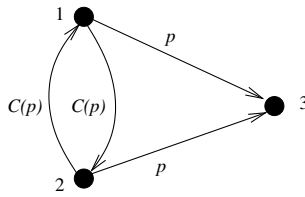


Figure 1. If two parties are to guarantee to cooperate in providing the same level of service to a third party, they must make bilateral cooperative promises to one another.

Promise type	Notation	Interpretation
Basic	$n_1 \xrightarrow{\pi} n_2$	Provide service/flow
Cooperative	$n_1 \xrightarrow{C(\pi)} n_2$	Imitate/Cooperate
Use	$n_1 \xrightarrow{U(\pi)} n_2$	Use/Accept from
Conditional	$n_1 \xrightarrow{\pi_1/\pi_2} n_2$	Promise 'queue': π_1 if π_2

Basic service promises form any number of types, e.g. 'provide web service in less than 5 seconds' or 'tell you the domain name of IP address'. A 'use' promise is the promise an agent makes to say that it will utilize a service provided by another agent. Just because a service is promised and provided does not imply that it will be used. Use promises can be important in offering assurance to a third party when one promise is made conditionally on another.

Cooperative promises are used in order to build consensus amongst individual agents with respect to a basic promise type. If π is a promise type, then $C(\pi)$ is a corresponding promise type between agents to collaborate or imitate one another's behavior with respect to π (see fig 1).

A promise is said to be *broken* if an agent makes two different promises of the same type at the same time (this is different from a promise that has expired or been changed).

3.1 Roles from promises

The graphical structure of promises allows us to define naturally occurring *roles* for hosts in a collaborative network, based on whether a node or group of nodes is a sender or a receiver of a particular type of promise: i.e. those nodes that point (or are pointed to) in a particular way have similar roles. See the example in fig. 2. In this figure one can identify several roles. Nodes 1,5,6,7 are all promisers of web service. This labels them with a role, which we might call 'web server'. Similarly, all the clients in the ring are receivers of the web service – they have a appointed role which we could call 'web client' (the servers point to them). We note that node 1's promise has a dependency, since it relies on a database service provided by a different agent. It confirms that it will make use of this with a use-promise.

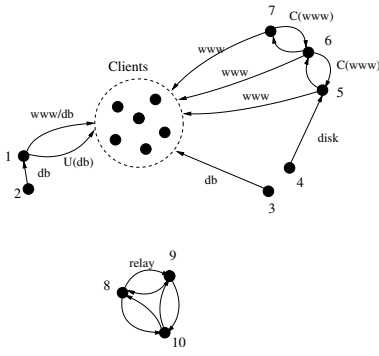


Figure 2. Example promise network illustrating roles.

3.2 Virtual systems and machines

Where are the boundaries of the system? We now have a model that allows us to consider everything from individual promises (implemented by individual server processes on a single host) to multiple promises that span many hosts.

We now see how promises can be applied to a virtual hosting scenario. It does not matter where agents are located, either virtually or geographically — what is important, in functional terms, is the role they play in a network.

Agents can be separated out as virtual machines, one per service role, in order to minimize dependencies or for convenient partitioning. Similarly agents in quite different physical locations can be understood together as backup or fail-over servers in a redundant scenario. They might belong to different companies, sharing a service agreement; similarly, the database server (node 3) and the web server (node 7) might be two virtual instances, running on the same physical machine.

Here, we discover another reason for this: a strategy of one virtual machine per service promise allows us to encapsulate and therefore easily manage promise roles individually.

Instead of deploying a single node with all services, a configuration can be crafted which supports many nodes each with a separate role. This leads to a cleaner set of services, which can be more robust. In addition, multiple nodes can be configured to service the same role. In this way, the set of services can be scalable, expanding beyond the boundaries of a single machine. Thereby adding more robustness to the configuration.

4 Managing infra-structure

Let us consider an implementation using virtual Linux (e.g. User Mode Linux or Xen) systems as a testbed for creating autonomous nodes. Autonomous management is easily implementable using the well-known configuration

service Cfengine. Cfengine is designed with an autonomous viewpoint of management and fits our model well. As an example, a shared configuration repository cannot push configurations onto the agents, but must let the agents pull it at their own will.

It is important to have a simple framework which preserves the autonomy, but which allows multiple nodes to be instantiated on a single hardware devices when that is desirable. Cfengine’s overlapping class model of virtual systems is ideally suited to this.

Multi-agent systems offer several approaches to management schemes, delegation policies and cooperation models. The choice of mechanism depends on the goals to be achieved, but there are several hybrid models being explored. For example, Mackin et. al [2] explore letting agents develop their own communication protocol that will suit them best, while Berghoff et. al [3] suggest a distributed management model where management is carried out by localized agents that report back to a centralized manager. However, most of these systems assume a single organization and policy structure. In a single organization environment, it is possible to strictly delegate system configuration in policy or to use these hybrid solutions. These mechanisms fail under the pressure of a multi-organizational environment. As the number of organizations increase, a market based solution becomes more attractive.

Service providers need to balance the needs for features and the needs for security for their many customers. Often, the process of managing system security translates into reduced functionality and less flexibility. This is an outgrowth of sharing the same resources. If a separate physical computer for each customer is used, it can increase security and increase features, but it leads to increased cost and fragmentation of resources. Security and configuration concerns can be addressed for customers by deploying services in a virtual server environment. These customized virtual machines can act as wrappers in separating independent users from one another, giving increased flexibility for user management, access and configuration.[4, 5] This virtual server scenario allows a service to be expanded beyond the boundaries of a single machine, adding more robustness and higher quality services in the long run.

5 A strategy of minimal encapsulation

Suppose there is a network of virtual machine instances and a set of defined roles. The responsibility rests with each virtual machine instance to implement the promises that are part of the role it has been assigned. For example, if an instance gets assigned the role “webserver”, it would not only install the appropriate software, but also configure it accordingly and make sure the webserver software keeps running. This is more than a software package management

system, it can be viewed as a subset of the entire configuration grouped together in the form of a role.

Some roles promisers depend on the services provided by other roles. For example, the “Webserver” depends on the “nameserver” to provide the name of the machine. The “Webserver” also depends on the “gateway” to provide a hole in the virtual network firewall in order to provide access to the web server from the outside network. This means that each agent promising a role needs to be context aware about the roles promised by the rest of the group in order to resolve dependencies.

These dependencies between roles can be seen directly from the promise graph. Roles that depend on the successful implementation of another role are pointed to by the role they depend on. Cycles and possible deadlocks can be identified easily by traversing the graph.

As long as the group as a whole keeps track of the assigned roles, virtual machine containers can come and go. The initialization time of a virtual host can be lower then adding a new physical machine to the network. This leads naturally into utility grid computing where an array of physical hosts lease resources and provide new virtual machines (see Fig. 2)

Using this approach, failing virtual machines may be removed and the roles they played transferred to other virtual machines. Using the same mechanism, small virtual machines can be traded in for larger virtual machines. Alternatively, the number of nodes serving a particular role can dynamically be increased or decrease as needed. This opens up for adaptation both internally on each virtual machine as it promises a new role, but also for the group that has to cooperate in order to achieve a global promise.

5.1 Configuration model

In the ISP-service-on-demand setting, a customer can choose what functionality she wants. Each choice will be added to the configuration database and implemented automatically. Hosting cost can be established based on the virtual networks resource consumption.

6 Implementation

This implementation demonstrates the core components of a decentralized, distributed fabric where roles can be created and managed. We assume a cluster of physical servers that are meant to take on roles for customers. Together they form an overlay where virtual machines can interact and realize the intended functionality of the cluster. Our implementation is based on a three layered design:

- **Physical Service Layer**

At the lowest layer is the physical cluster. These physical machines have only one purpose, and that is to run

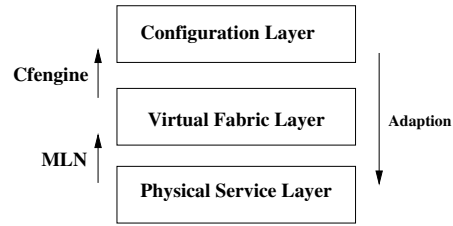


Figure 3. Configuration management is divided into three layers, where the top layer is responsible for role assignment.

virtual machines. Xen is a virtualization mechanism where virtual machines can be run with very low overhead. [6] This low overhead makes Xen attractive for use in production environments.

- **Virtual Fabric Layer**

MLN provides a management layer for virtual machine hosting [7]. It provides a management interface to logical groups of many virtual machines that may span several physical servers. The network of virtual machines create a overlay fabric called the *virtual fabric layer*. In MLN, networks of virtual machines are grouped into projects. A project is described with a declarative configuration language. Third parties can use MLN query commands to access the data structure. The data belonging to a project can be modified via MLN commands enabling an expert system to modify the configuration of a project automatically. Important aspects of MLN are the ability to upgrade running projects and a client/server architecture that lets one manage projects that span several physical machines atomically. The syntax of the project file can be generically expanded giving room for third parties to store their own data as well.

- **Configuration Layer**

This layer consist of the expert systems in place inside the virtual machine that enables it to grow into a role. The expert system has access to the management interface of MLN and can therefore enforce changes to the MLN project it is itself a part of. Modifications to the project could be the addition or removal of virtual machines as well as defining on what physical host to run on. Other modifications can occur on a higher level, such as transitions of roles between cfengine agents.

This top layer is also where the promises are represented. The promises can, as we noted above, be translated into host configuration tasks that has to be autonomically handled by the individual virtual machine. The automated deployment of large-scale computing infrastructure and con-

figuration has been an active research project for more than a decade. Significant progress has been made in issues such as uniformity, predictability and customization [8, 9]. Cfengine fits into the task of managing *role* configuration and performance [10]. Cfengine can perform actions based on boolean triggers which can stem from system events or variable properties. This reactive behavioral model is sufficient in order to implement an expert system that can accomplish basic self-managing behavior.

Cfengine implements a *convergent management* model which tries to move the configuration back into its intended state should it be modified in some way. Convergence acts as a form of configuration regulation that allows hosts to be managed independently of any central authority. Convergent management leads a system into a specific configuration and then maintains it there over time [11, 12].

6.1 An ISP web hotel

The goal of the ISP web hotel is to host web services for various customers. It consists of a gateway with access to a back-end of server nodes. Once a new web service function for a customer is created, an MLN project for that customer is written. The MLN project defines the starting point (See Fig. 4). A role is assigned to a server by placing a virtual machine with that role on it. The gateway server will therefore play a gateway for each customer. One or more backend servers can play a role as a webserver for that particular customer. The gateway role must route incoming web requests to these roles distributing the load between them. This leads to another role for the gateway - that of a load balancer. The topology is shown in Fig. 5. Although the topology in the diagram may give the impression of centralized control, it is not the case. The design is for traffic flow purposes, so that traffic flows through a gateway role and to the back-end cluster of webserver roles. Every role is autonomous.

For each *role type*, there is a cfengine configuration description which includes lists of promises (see Fig. 6.) These promises are concrete tasks that cfengine can handle autonomously such as software that must be installed, changes to system configuration files and information that must be shared with other virtual machine instances about the role.

The data structure and API from MLN is used by cfengine to extract information about the given node it runs on. Every virtual machine will get a copy of the project file and the MLN tool internally, which they use for low-level management.

The entire collection of promises towards a customer will configure itself from the ground up automatically according to the roles that have been assigned. As a virtual machine is initiated as a new container for a role, it is started

```
global {
  project p1
}

superclass host {
  size 2500M
  memory 128M

  network eth0 {
    switch lan
    netmask 255.255.255.0
  }
}

host one {
  superclass host
  service_host backend1
  roles {
    webserver(mysite.com)
  }
}

host two {
  superclass host
  service_host gw_server
  roles {
    gateway
  }
}
```

Figure 4. A virtual network written in the MLN language.

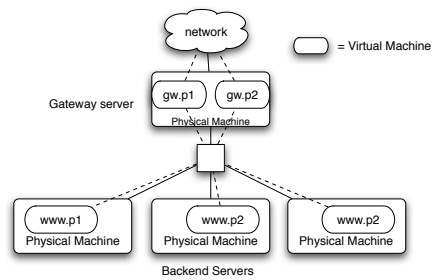


Figure 5. The cluster testbed on which the virtual sites run.

by MLN from a basic configuration that is then later specialized by cfengine inside the virtual machine itself. Such specialization can be the addition of another network interface for a gateway role or installing webserver software. Every role encapsulates all the promises defined by the role. A physical machine may have roles that regard different customers. These roles will not interfere with each other because of their encapsulation.

7 Discussion

Can all the aspects of a role or intended system function be covered by a set of simple and implementable promises? Let us answer this with a counter example. Given that one has a host system with a desired function P . In order to manage the system into this function one needs to know all necessary promises that lead to P . If not, the function will never be realized properly. If it therefore turns out to be difficult to define all the promises, it is a clear indication

```

control:
    enabledRoles = (
        ExecShellResult('`mln parse -f /var/vs.mln
                        -S /host/$host/roles`' )
    )
classes:
    lgateway::
        environment = ( PrepModule('`mln parse -f /var/vs.mln
                                    -S /global/vs/gateway.address`' ) )
processes:
    WebServer.iHaveOutsideLink::
        "httpd" restart "/usr/bin/httpd"
shellcommands:
    gateway::
        "/bin/echo 1 > /proc/sys/net/ipv4/ip_forward"

```

Figure 6. An excerpt from a cfengine configuration showing how roles are represented in terms of classes.

that the intended role is not well designed. Cfengine provided our implementation the necessary means to define the details of each promise in each role in our scenario. One should keep in mind that if a particular tool is not capable of implementing a role properly it might be that another tool could do it in its stead.

The highest cost in terms of resources is the allocation of memory for each virtual machine. A role should therefore incorporate some mechanisms to scale itself down in terms of resource demands if it is less used. The foundation for this self-modification is already in place.

A promise can be moved to another server transparently using live migration of virtual machines. For this to work the servers would have to have concurrent access to the virtual machines filesystem.

8 Conclusions

We have demonstrated a prototype for the encapsulation of system boundaries into self-managing virtual machines. The role of each virtual machine can be justified clearly through the Promise Theory of autonomous agents.

This paper describes and implements a model for dynamically reconfigurable business environments from an abstract blueprint. The prototype framework demonstrates how modern operating systems can be given the ability to manage their system resources in an autonomic manner, without continuous human interaction.

This Virtual Site model creates a foundation for exploring additional components of multi-agent systems, such as negotiation of agreements through voluntary cooperation. For consumers, one can envisage a scheme for receiving customer requests directly and deploying virtual instances through a rewriting engine.

References

- [1] Mark Burgess, "An approach to policy based on autonomy and voluntary cooperation," *Lecture Notes on*

Computer Science, vol. 3775, pp. 97–108, 2005.

- [2] Kenneth J. Mackin and Eiichiro Tazaki, "Evolving intelligent multiagent systems using unsupervised agent communication and behavior training," *Systems, Man, and Cybernetics, 2000 IEEE International Conference*, 2000.
- [3] Anselm Lignau Jürgen Berghoff, Oswald Drobnik and Christian Mönch, "Agent-based configuration management of distributed applications," *Configurable Distributed Systems, 1996. Proceedings.*, pp. 52 – 59, 1996.
- [4] Peter A. Dinda Renato J. Figueiredo and Jos A. B. Fortes, "A case for grid computing on virtual machines," *Proceedings. 23rd International Conference on Distributed Computing Systems.*, 2003.
- [5] C. Sapuntzakis and M. S. Lam, "Virtual appliances in the collective: A road to hassle-free computing," in *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HOTOS IX)*, 2003.
- [6] Paul R. Barham, Boris Dragovic, Keir A. Fraser, Steven M. Hand, Timothy L. Harris, Alex C. Ho, Evangelos Kotsovinos, Anil V.S. Madhavapeddy, Rolf Neugebauer, Ian A. Pratt, and Andrew K. Warfield, "Xen 2002," Tech. Rep., University of Cambridge, Computer Laboratory, Jan. 2003.
- [7] K. Begnum, "Mln website," <http://mln.sourceforge.net>, WWW.
- [8] M. Burgess, "A site configuration engine," *Computing systems (MIT Press: Cambridge MA)*, vol. 8, pp. 309, 1995.
- [9] S. Traugott and J. Huddleston, "Bootstrapping an infrastructure," *Proceedings of the Twelfth Systems Administration Conference (LISA XII) (USENIX Association: Berkeley, CA)*, p. 181, 1998.
- [10] M. Burgess, "Cfengine www site," <http://www.iu.hio.no/cfengine>, 1993.
- [11] M. Burgess, "Configurable immunity model of evolving configuration management," *Science of Computer Programming*, vol. 51, pp. 197, 2004.
- [12] A. Couch and Y. Sun, "On observed reproducibility in network configuration management," *Science of Computer Programming*, vol. 53, pp. 215–253, 2004.