Distributed System Administration Technologies a State of the Art

Jakub Zwolakowski

Distributed System Administration Technologies a State of the Art [1]

Jakub Zwolakowski

September 2012

[1] Work partially supported by Agence Nationale de la Recheche, through the Aeolus project, grant number ANR-2010-SEGI-013-01, and performed at IRILL, http://www.irill.org, center for Free Software Research and Innovation in Paris, France.

Contents

Chapter 1

Introduction

The term *distributed configuration management* refers to the problem of managing the configuration of software components on multiple hosts, which are usually working together in a computer network. In most cases this task is pursued by *system administrators*, who are responsible for planning and carrying out all the required modifications: install, remove, upgrade and configure software. Traditionally they perform these operations "by hand", accessing the machines either directly or through the network.

When the demand for computing resources increases, be it in the case of simple workstations or expensive web servers, the number of hosts to manage grows. With them, the number of software components and the number of their relationships and dependencies raises as well.

The resulting complexity problem has been partially offset by the progress of technology. Constantly augmenting capabilities of hardware and adoption of new solutions, like virtualization and cloud computing, make it easier and less expensive to create big networks of machines. But the work of a qualified system administrator is not as easily scaled up. Hence the popularity of solutions aimed to help system administrators to handle the resulting complexity.

A *distributed configuration management tool* is a piece of software, which helps a system administrator to:

- define, store and maintain reusable configuration artifacts,
- automate the process of enforcing configuration changes on multiple hosts at a time.

A basic improvised tool of this kind could be as simple as a shell script, that is used to execute the same administration command on a set of remote machines, which are reachable via SSH.

Modern distributed configuration management tools are far superior to this example and offer much more both in terms of functionalities and efficiency. Most of them features some kind of a specialized abstract language used to describe configuration, offers a centralized and structured way to store configuration descriptions, and allows to easily define, which parts of the global configuration are relevant for which groups of machines. Moreover, such configuration management tools take care of the process of deploying the desired configuration on the hosts without requiring per-machine human interaction.

Typically in these advanced tools all the configuration information is stored on the central server and distributed between hosts not by *pushing* it from server to hosts, but rather by so-called *pulling*. Each host queries the server periodically about its new desired configuration (and/or how to attain it) and then it applies required changes locally. This way whole system is more scalable, as some part of work is shifted form the configuration server to the hosts.

An important concern associated with distributed configuration management is *service orchestration*. This broad term contains all the aspects of automated coordination and management of services dispersed between multiple hosts. As services running on different machines participate in implementing a single coherent service relevant for end-users, each service should be changed in accordance with others. For instance, a depended upon service should not be shutdown (e.g. for

upgrade purposes) before the services that depend on it, and vice-verse for its start-up (e.g. at the end of its upgrade). Proper coordination is not an easy task, as services not only depend on each other and cooperate in various ways, but also depend on a whole range of software components that should be locally installed on the machines where the services run. Modern distributed configuration management tools offer a varying degree of service orchestration capabilities.

Relevance to the Aeolus project

The ANR funded Aeolus project aims to tackle the scientific problems that need to be solved to bridge the gap between Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) solutions, by developing theory and tools to automate deployment, reconfiguration, and upgrades of variable sized, non-homogeneous machine pools

One of the important axes of Aeolus is the development of a *formal model* of complex distributed component-based software systems that, as it happens with the so called "clouds", can change in size over time due to the addition and removal of new (virtual) hosts. The formal modeling of such systems is then intended to be used to automatically plan software reconfigurations starting from high-level request submitted by system administrators. To achieve that goal, the formal modeling should not be separate from the reality of existing technologies. It would indeed be pointless to have a very expressive formal model, if instances of it cannot faithfully represent the reality of actual clouds, or local networks, for the purposes of planning software upgrades or other reconfigurations.

The purpose of this study is then to review and compare the state of the art of technologies for distributed configuration management. In particular, we focus on those technologies that are applicable to deploy FOSS (Free and Open Source Software) components on UNIX-like machines, as they constitute the technological *milieu* of interest for Aeolus. The review and comparison is meant to be useful for Aeolus for several reasons:

- Aeolus aims at developing real and useful technology. To that end we need to start from what is used today by system administrators and ensure that *present* use cases, that existing technologies have been developed to solve, can be grasped by the formal models Aeolus is going to develop. During this study we have also found recurrent concepts that exist throughout all reviewed technologies. They are likely to form some sort of informal "ontology" that corresponds to the way of thinking of system administrators. We plan to use such knowledge as guidance for the development of Aeolus formal models.

- During the lifetime of Aeolus, we will also need to test the planners that will be developed as part of the project on real data and compare the outcome of planners with what system administrators would have done "by hand". Knowing existing technology will allow to evaluate the feasibility of obtaining test data as translation from languages supported by existing tools to Aeolus-specific languages that will be developed by Aeolus partners.

- On the other end of the spectrum, it will be useful to reuse existing technologies as low-level deployment tools. If that will turn out to be feasible, one can imagine fruitful pipelines where Aeolus planners are used to solve complicate re-configuration headaches (e.g. major upgrades of a FOSS distribution on all the machines that constitute a private cloud), and existing distributed configuration management tools are used only for the actual deployment of configuration changes.

Work done

We have chosen the most popular distributed configuration management tools available on the FOSS market for UNIX-like machines, and we have studied their functionality and architectures. The work has been based mostly on the available documentation for the tools and on already existing examples of their use, like provided patterns and solutions. In certain cases we also applied some amount of practical experimentation.

The purpose of this research can then be summarized as:

- To better understand the problem of distributed configuration management, by analyzing existent working solutions.
- To extract the *models* used, explicitly or implicitly, by each tool and grasp the similarities and differences between them.
- To spot interesting mechanisms, methods and solutions of subproblems employed in the researched tools.
- To examine and appraise the tools in context of potential application in the *Aeolus Project* as the low-level deployment layer.

We have considered the following distributed configuration management technologies:

1. CFEngine3 (by *CFEngine*) [1]
   Third version of one of the first true distributed configuration management tools. Flexible, versatile and robust, but lacking the focus and ease of use of its competitors. It works on a rather low level of abstraction. We can find many basic CFEngine3 ideas and solutions reused in other tools.
2. Puppet (by *Puppet Labs*) and MCollective [2]
   A very popular tool. Inspired heavily by CFEngine3. It models the configuration of managed hosts by introducing the concept of abstract resource layer. Also it features a high focus on declarativity. When extended with MCollective, it gains a broader perspective and some capabilities of service orchestration.
   (Note: "Puppet alone" and "Puppet with MCollective" will be considered as two separate tools. Puppet on it's own is similar in its capabilities to CFEngine3, whereas when coupled with MCollective, it becomes more or less equivalent to Chef.)
3. Chef (by *Opscode*) [3]
   In a nutshell, Chef is Puppet reworked from another point of view, abandoning the principle of declarativity and giving more direct control to the administrator. Also, it introduces several interesting mechanisms permitting a certain level of service orchestration.
4. juju (by *Canonical*) [4]
   An entirely different approach to the distributed configuration management, dedicated completely to cloud environments. In fact it is much more a service orchestration framework, than configuration management tool. juju is highly focused on the services and their relations and neglects many aspects of configuration management which fall below this level of abstraction. Moreover, its desire to retain a simple and clean model makes it simplistic and restrained.

1.1 Methodology

In order to show the differences and similarities between the inquired tools clearly, they will be presented in a specific order. The chosen order makes it easier to follow the variation in ideas (and in implementations of these ideas) between the presented tools in a systematic and logical way. In most cases it will be sufficient to mention explicitly only the differences between each tool and the previous one in the sequence:

1. CFEngine3
2. Puppet
3. Puppet with MCollective

4. Chef
5. juju

We have discerned several important and interesting aspects concerning functionality and architecture of the distributed configuration management software. In order to examine the tools objectively and effectively, we have organized these aspects into five main axes of comparison:

1. **Domain:**
   Which objects and relations can the tool manage?
   How are they modeled and structured?
2. **Language:**
   What are the characteristics of the language used in the tool for describing the configuration and/or describing the way to attain a given configuration?
3. **Automation:**
   Which useful mechanisms and solutions does the tool employ?
   Can we find any valuable and interesting ideas among them?
4. **Architecture:**
   What is the structure and organization of the tool itself?
   What is the structure of the distributed environment that the tool can manage?
5. **Platforms:**
   Which operating systems can the tool support?
   How well does it handle the problem of heterogeneity of platforms?

One important thing, which we did not take into consideration studying the tools and preparing this comparison, is their capabilities of automatic cooperation with the Cloud Provider. It is worth mentioning, that some of them have quite sophisticated modules dedicated to this role and are able to integrate the dynamic aspect of cloud computing into their framework (e.g. they can automatically instantiate new machines).

Axes of comparison

Domain All the configuration management tools are, by definition, designed to manage the configuration of some software components. We define the whole area which can be controlled by a tool as its domain and all the different objects controlled by the tool (existing inside the domain) as the resources.

1. What can the tool control? Where are the borders of his domain?
2. What is the catalog of available resources?
3. How precise control do we have over each resource's state and behavior?
4. Do possibilities of defining new types of resources exist?
5. Can we express relations (like dependency or conflict) between the resources?
6. What notions of location of resources exist? Can we distinguish between the resources on different machines?

Language Every configuration management tool needs to allow the system administrator to express the desired state of the system and/or describe actions needed to attain it. All of the compared tools use some kind of language to either specify the desired configuration itself or to write configuration scripts (which are executed in order to arrive at the desired system configuration).

1. Is the tool using a special DSL (Domain Specific Language) or a generic programming language?
2. What is the expressivity of the language? Does it impose important constraints on what we can do and describe?
3. What is the level of declarativity of the language?
4. Is the language used to specify the resources themselves or rather to indicate actions performed on the resources?
5. How naturally can we express our ideas using the language? Is it well adapted to its tasks?

Automation The very reason of using a distributed configuration management tool is to automatize tasks associated with deploying and maintaining the software components. Therefore the issue of practical benefits provided by a given tool seems to be crucial in order to assess its value.

1. What do we gain using the tool over what we can do by hand?
2. Are any advanced algorithms or solvers used?
3. What is the level of verification of correctness and feasibility of our decisions and actions?
4. Is the tool designed to handle the distributed configuration management and/or also orchestration of distributed services?

Architecture The structure of the tool itself is tied with the architecture (or possible architectures) of the environment, which that the tool is designed to manage.

1. What is the top-level schema of the tool's architecture?
2. What variety of the environment's structures can the tool handle?
3. Is the tool itself distributed or centralized? To what degree?
4. What is the configuration propagation method:
    - *push* (we actively enforce the desired configuration on every machine)
    - or *pull* (every machine asks for its desired configuration by itself)?

Platforms Different tools are designed to work with a different variety of platforms and operating systems.

1. Which platforms are supported by the tool?
2. What level of infrastructure's heterogeneity is easily handled?
3. How automatic is heterogeneity handling? How big part of these differences in configuration, which result purely form differences between the platforms, do we have to explicit?

1.2 Deployment example

In order to ease the comparison, we will employ a common deployment example throughout all technology presentations. As the deployment example, we will use a quite standard LAMP (Linux, Apache, MySQL, PHP) configuration with Wordpress (a very popular blogging tool and Content Management System), deployed on two separate machines (deployment diagram, including relationships between the components, is depicted on Figure 1.2):

- Machine 1: **Application Server**
  - Linux : operating system
  - Apache2 : web server
  - PHP5 : script language
  - Wordpress : application
- Machine 2: **Database Server**
  - Linux : operating system
  - MySQL : database management system

We will try to show, how this example can be implemented using all the compared tools. The proposed implementations are slightly simplified in order to hide unnecessary details. Therefore, they should be viewed only as illustrations of concepts (though they are very close to the final code which could be really used in the presented tools).

This exact example was chosen for three main reasons:

- It is complex enough to show most of the mechanisms which interest us. It does not only require employing pure distributed configuration management techniques, but also leaves place for some amount of service orchestration, as we need to establish the connection between two working services deployed on separate machines: the Wordpress application and the database.
- At the same time it is simple and clear enough to be easily comprehensible. Its implementations in all the tools are reasonably short.
- Moreover, it is very typical and common. It features only very well known and widely used technologies.

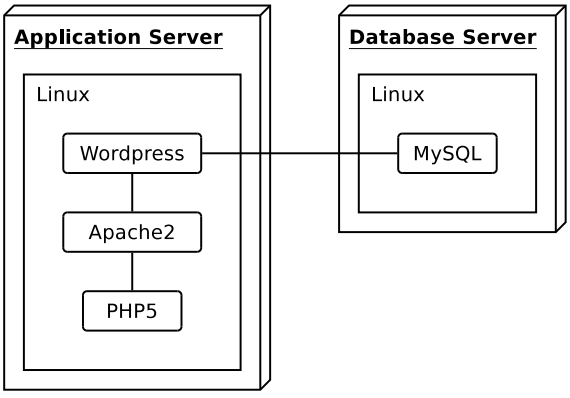


Figure 1.1: A coarse deployment diagram of the LAMP example.

1.3 Document structure

This document is split into three main parts.

- Chapters from Chapter 2 to Chapter 6 present the tools one by one, summarily, following the guidelines that have been discussed in the previous section.
- Chapter 7 puts all technologies together and arranges them in a tabular comparison that sums up the state of the art in distributed configuration management tools.
- More tool details, not strictly necessary for tool comparison, but useful to evaluate technology applicability in the context of Aeolus, are collected in Appendix A.

Chapter 2

CFEngine3

CFEngine was one of the first modern distributed configuration management tools, created in 1993 by Mark Burgess. Some years later its author developed the *Promise Theory* and in 2008 he applied it constructively in the third version of his tool: CFEngine3.

Particularities:

- Very flexible and versatile. CFEngine3 allows us not only to manage the configuration of the resources on the hosts, but also to control how the configuration data itself is transferred between the hosts and applied.
- Capable of coping with unreliable infrastructure and therefore it is well adapted to so-called "mission-critical operations".

2.1 Domain

2.1.1 What can we control?

Each machine can make promises about its resources, regarding their state and behavior. The catalog of available resources is thus implied by the available promise types. We cannot define new promise types, but the CFEngine3's repertoire is very rich, as each promise type comes with with a large amount of options and parameters.

Main available promise types:

- FILES: files and directories in the filesystem
- PACKAGES: software packages
- COMMANDS: arbitrary shell commands
- PROCESSES: running programs
- SERVICES: a set of processes and commands bundled together
- INTERFACES: network interfaces
- STORAGE: disks and filesystems
- DATABASES: tables in databases (SQL, LDAP, MS Registry)
- ENVIRONMENTS: enclosed computing environments (physical machines, virtual machines, clouds)

2.1.2 Relations

CFEngine3 does not offer us a full-fledged system for defining relationships between promises. With some effort it is effectively possible to force two promises (in one machine's scope) to depend on each other explicitly, but the mechanism permitting us to do so is quite artificial and cumbersome. However, we can easily document dependencies between promises for the sake of knowledge management.

In order to avoid basic dependency problems, which appear naturally when we are executing sequences of promises, CFEngine3 employs a mechanism called *Normal Ordering*. It consists of a set of simple rules, which decide about the precedence of execution of the promises in the sequence, trying to minimize the possibility of failure caused by dependency issues. For example, as services often depend on software packages, in *Normal Ordering* we execute all the promises about packages before moving on to promises concerning services.

CFEngine3 provides us however with one (and only one) type of promises which support dependencies natively: promises about services. If we declare, that service X depends on the service Y, and we order CFEngine3 to start X, service Y will be automatically started before. We can thus explicitly define the required order of starting services.

2.1.3 Location

Promises exist always only in the scope local to a single machine. Each machine is completely autonomous in choosing its promises. However, by default it makes a promise to synchronize all its promises with the configuration server.

CFEngine3 has some *Distributed Discovery* capabilities, which let us condition the execution of certain promises on the state and behavior of other machines in the environment. We accomplish this using a set of built-in *environment-probing functions*. For example, the `selectservers` function takes a list of hosts and finds out, which of them respond correctly to a specific request sent over the network.

2.2 Language

CFEngine3 features its own *Domain Specific Language* for defining promises. This promise syntax is completely declarative, even though it features a specific understanding of declarativity. In fact, when we write CFEngine3 promises, we do not really describe directly the desired *state* of the resources. We rather provide a set of *actions*, which should be performed on the resources in order to put them in the desired state.

CFEngine3's language is highly expressive and powerful, although sometimes very unwieldy. It follows the principle of declarativity so zealously, that it even introduces special types of promises in order to implement some of the language primitives in a declarative way (like variables and conditions).

2.3 Automatization

In fact, the promises that we write are equivalent to parameterizable high-level scripts, built from preprepared blocks and executed by the CFEngine3 framework. Nevertheless, the promise syntax offers us a significant level of abstraction and modularity, as well as several built-in mechanisms permitting to infuse the configuration data with commentary and documentation.

Hence, CFEngine3 offers us two basic gains over doing things "by hand":

- Knowledge management: we can easily maintain and reuse pieces of configuration (i.e. promises), because they are abstract, modular and well documented.
- Execution framework: thanks to CFEngine3 we do not have to execute any scripts ourselves. Instead we write them in the form of promises and we put them in the right places

of the framework. Then CFEngine3 takes care of all the execution details (including some quite advanced details, like transferring files between hosts, etc).

Other than that, CFEngine3 does not incorporate any interesting (from our point of view) solutions. Most of the mechanisms used in it are quite straightforward and not very sophisticated. There is almost no verification or checking of our decisions. Also, CFEngine3 does not posses any service orchestration capabilities worth mentioning.

2.4 Architecture

CFEngine3 is very flexible and works in a truly distributed fashion, all of its parts are quite autonomous by design. As it does not really have a fixed architecture structure, it can be adapted to work reliably in various environments. However, in almost all normal cases we just want a typical centralized architecture, which is implemented by the default scaffolding of promises.

Configuration data is propagated by CFEngine3 using the pull method, albeit we can simulate a push of changes if need arises.

2.5 Platforms

CFEngine3 supports all the common platforms: *nix, Mac, Windows. However, serious support for Windows is available only in the commercial version.

In theory, CFEngine3 can cope without trouble with any level of heterogeneity in our infrastructure. Though achieving that in practice is much more complicated, as we have to explicitly handle almost every little difference between the description of configuration on various platforms (for example, even on the level of file paths CFEngine3 does not automatically convert slashes to backslashes between UNIX paths and Windows paths).

2.6 Example

2.6.1 Promises

For the purpose of this example, we will just declare all promises in a single file: `mypromisses.cf`. In reality, making these promises work would require a few more steps. For example, we would have to modify the main promise file to make CFEngine3 actually include and execute promises declared here.

Promise file: mypromisses.cf

```
# Common

bundle common globals {

variables:
  "dbName"     string => "wordpress-db";
  "dbUser"     string => "wordpress-db-user";
  "dbPassword" string => "wordpress-db-password";

}
```

First, we have declared some variables, which will be used on our both machines. The form of this declaration is quite unusual and very specific to CFEngine3: each of the three central lines above is in fact a single promise. In this context, we should read them as: "I promise, that a global variable X will have a string value of Y" (*a global variable*, because they are all declared in the bundle `globals`, which makes them available in all other bundles).

```
# Wordpress

bundle agent wordpress-server {

variables:

  "wordpress-packages"
    slist => {
      "apache2",
      "php5",
      "libapache2-mod-php5",
      "mysql-client",
      "php5-mysql",
      "wordpress"
    };

  "siteName"   string => "mySite";
  "configPath" string => "/etc/wordpress/config-$(siteName).php";

  "sitePath"   string => "/var/www/$(siteName)";
  "siteSource" string => "/usr/share/wordpress";

  "hostname"   string => "app-server";
  "dbHost"     string => "db-server";


files:

  "$(sitePath)"
    link_from => linkdetails("$(siteSource)");

  "/etc/apache2/sites-enabled/000-default"
    create => "true",
    edit_line => expand_template("wordpress-vhost");

  "$(configPath)"
    create => "true",
    edit_line => expand_template("wordpress-config");

packages:

  "$(wordpress-packages)"
    package_policy => "add";

services:

  "apache2"
    service_policy => "start";

}

body link_from linkdetails(tofile) {
  source        => "$(tofile)";
  link_type     => "symlink";
}
```

We proceed to declare the promises for the application server, grouping them in a single bundle `wordpress-server`. As declarations concerning all the software components (PHP, Apache2 and Wordpress) are mixed together, the code looks a little unorganized. CFEngine3's agent will take

care of executing the promises in a preprogrammed order (it will install software packages before starting services, etc.), which in this case is sensible and sufficient.

We can spot several interesting CFEngine3 idiosyncrasies in this piece of code. For example, we can see how a list of packages, assigned at the beginning to the variable `wordpress-packages`, is automatically executed in a for loop: in the section `packages` the list will be expanded and each element will be used to create one promise. We can also see, how a structure called body attachment (`body link_from linkdetails`) is used in the declaration of a symbolic link. This is not a smart method to modularize the code: we just need to declare a nested attribute (attributes `source` and `link_type` are in fact components of the `link_from` attribute) and there simply is no other way to do it in CFEngine3.

```
# MySQL

bundle agent mysql-server {

variables:

  "createDbScriptPath" string => "/tmp/create-db.sql";

classes:

  "databaseExists"
    expression => returnszero("mysql -uroot -e \"use ${dbName}\"");

files:

  "$(createDbScriptPath)"
    create => "true",
    edit_line => expand_template("create-db.sql");

packages:

  "mysql-server"
    package_policy => "add";

services:

  "mysql"
    service_policy => "start";

commands:

  !databaseExists::
    "mysql -uroot < $(createDbScriptPath)";

}
```

In the declarations of promises for the database server we can see another peculiar CFEngine3 mechanism: conditions taking form of so-called classes. The class `databaseExists` in fact embodies a condition, which is true only when the database already exists (more precisely, when the shell command designed to check the existence of the database returns zero). The promise creating the database (declared in the section `commands`) will be executed only if this condition is false.

2.6.2 Templates

Template file: wordpress-config

```
/** The name of the database for WordPress */
```

```
define('DB_NAME', '$(dbName)');

/** MySQL database username */
define('DB_USER', '$(dbUser)');

/** MySQL database password */
define('DB_PASSWORD', '$(dbPassword)');

/** MySQL hostname */
define('DB_HOST', '$(dbHost)');
```

Template file: wordpress-vhost

```
<VirtualHost *:80>
  ServerName $(hostname)
  DocumentRoot $(sitePath)

  <Directory $(sitePath)>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    allow from all
  </Directory>

</VirtualHost>
```

Template file: create-db.sql

```
CREATE DATABASE $(dbName);

GRANT ALL PRIVILEGES
ON $(dbName)*
TO '$(dbUser)'
IDENTIFIED BY '$(dbPassword)';
```

As we can see, the templates mechanism in CFEngine3 is not complicated. The only thing that happens between reading the text of a template file and putting it into the given output file is a simple substitution of variables with values (according to variable bindings available in the promise execution context).

2.7 Summary

This is a synthetic summary of the CFEngine3's characteristics. Full summary, including all the tools, can be found in the table 7 on page 41. Description of the employed terminology is available in the table 7 on page 42.

Model	domain	resources (e.g. file, package, service)
	relations	none
	control mechanism	promises about resources' state and behavior
	hierarchy	flat model
Language	kind	own DSL
	style	declarative
	use	actions on resources
	expressivity	restrained (but quite high)
	pertinence	cumbersome, too declarative
Automatization	benefits	equivalent to configuration scripts execution framework
	configuration management	high
	orchestration	low
	requests	no
	solver	no
Architecture	architecture	distributed
	particularities	by default simulates centralized architecture
Platforms	supported	all
	handling heterogeneity	low

Table 2.1: CFEngine3 summary

Chapter 3

Puppet

PUPPET is a very popular configuration management tool created by *Puppet Labs* and written in *Ruby*.

Particularities:

- Completely model-driven and declarative.
- Introduces explicit relationships between the resources.

3.1 Domain

3.1.1 What can we control?

A model based on resources lets us control the configuration of each machine. Every resource's state corresponds directly to the state of a specific part of the machine's configuration. The mechanism called RAL (Resource Abstraction Layer) takes care of linking abstract resources with their platform-dependent implementations.

Catalog of resources

The three most important resource types (called *trifecta*):

- FILE: a file or directory in the filesystem
- PACKAGE: a software package
- SERVICE: a service in the UNIX sense of the term (e.g. ssh, apache2, ntp)

All the native resource types: augeas, computer, cron, exec, file, filebucket, group, host, interface, k5login, macauthorization, mailalias, maillist, mcx, mount, nagios, notify, package, resources, router, schedule, scheduled_task, selboolean, selmodule, service, ssh_authorized_key, sshkey, stage, tidy, user, vlan, yumrepo, zfs, zone, zpool

3.1.2 New types of Resources

In PUPPET we can introduce new types of resources in two ways:

Inside the model: We can define new types of resources by bundling together several resources of existing types. This way we can use basic elements available in the model to build complex configuration components, parameterizable and reusable.

Extending the model: By writing *Ruby* plugins, we can integrate desired parts of system configuration into the Puppet's model as new types of resources (hence effectively extending the model).

3.1.3 Relations

In Puppet we can declare explicit relationships between the resources in the scope of one machine. There are two types of relationships available:

- Dependency relationship: *"X depends on Y"*,
  which means that X requires Y to be deployed correctly.
- Refresh relationship: *"X refreshes when Y changes"*,
  where the meaning of "refresh" depends on a resource type. Typically we introduce this kind of relationship between services and their configuration files. In that case it means "restart the service X when its configuration file Y changes".

3.1.4 Location

In Puppet resources exist only in a scope local to a single machine. Therefore, we can define relationships only between resources living on the same machine, as other resources are simply not visible from this scope.

The central server is responsible for the crucial task of performing mapping from the list of machines to lists of resources (i.e. it knows, which resources are relevant to which machines), hence deciding about the configuration of each single machine. Existence of this centralized mechanism gives us the possibility to coordinate the configuration of several hosts (for example using some common variables), but it does not provide us with any true service orchestration capabilities.

Note: in Puppet there exists also a relatively artificial mechanism of so-called virtual resources. It allows us to share resources between multiple machines, thus permitting us to simulate relationships between resources which live on separate hosts. Unfortunately, this mechanism works in a rather irregular and dubious way. For instance, the synchronization of virtual resources between the hosts is quite unpredictable. Maybe virtual resources can be used with some success in practice, but they are certainly not fitting seamlessly with the Puppet's resource model. Therefore we do not consider them a natural addition to the model, but rather a workaround.

3.2 Language

We define resources using the Puppet's own *Domain Specific Language*, which is completely declarative. It does not have the full expressive power of a generic programming language, so although it is very well suited for typical applications, it may become quite cumbersome if we try to implement non-completely-standard ideas and solutions. But this is a typical characteristic of almost every specialized language and overally we consider, that Puppet's language is consistent and well adapted for its job.

3.3 Automatization

Thanks to a real resource-based model, in Puppet the only thing we need to do in order to enforce a certain configuration policy is describing the abstract resources and their relationships. After that, it is Puppet's job to converge the system configuration to the desired state.

Puppet incorporates some interesting solutions associated with manipulating configuration data. There exists a resource description compilation step: for each machine the list of resources is compiled to the form of a directed graph (where edges denote dependencies) before performing any actions. Moreover, in Puppet we have some verification mechanisms, helping to control the

sanity of our resource declarations. The most noticeable one is checking for dependence cycles in the graph of resources.

However, Puppet is still a tool focused completely on distributed configuration management and does not provide us with any service orchestration capabilities.

3.4 Architecture

Puppet features the standard centralized architecture with one server and multiple nodes. Configuration decisions are distributed by the pull method.

3.5 Platforms

Puppet supports all the common platforms: *nix, Mac and Windows. Handling the platform heterogeneity is automated up to some point: the mechanism of providers permits us to use a uniform resource declaration interface across all the platforms. Above that level we have to handle differences between the operating systems explicitly.

3.6 Example

3.6.1 Manifest

Manifest file: mymanifest.pp

```
# Common database configuration
$db-name     = "wordpress-db"
$db-user     = "wordpress-db-user"
$db-password = "wordpress-db-password"
```

First we declare some variables, which will be used on both machines: from one side (the database server) to configure the database itself; from the other side (the application server) to configure access to that database.

```
# Wordpress

class wordpress-server {

  # Apache2

  package {"apache2":
    ensure => present,
  }

  service {"apache2":
    ensure  => running,
    enable  => true,
    require => Package["apache2"],
  }

  # PHP5

  package {"php5":
    ensure => present,
  }

  package {"libapache2-mod-php5":
    requires => [ Package["php5"], Package["apache2"] ],
```

```
  ensure   => present,
  notify   => Service["apache2"],
}

file {"/etc/php.ini":
  ensure => file,
  notify => Service["apache2"],
}

# MySQL client + PHP support

package {"mysql-client":
  ensure => present,
}

package {"php5-mysql":
  ensure  => present,
  require => [ Package["php5"], Package["mysql-client"] ],
}

# Wordpress

package {"wordpress":
  ensure  => present,
  require => [ Package["apache2"], Package["php5-mysql"] ],
}

# Putting the site in place and adding a Virtual Host

$siteName   = "mySite"
$configPath = "/etc/wordpress/config-$siteName.php"

$sitePath   = "/var/www/$siteName"
$siteSource = "/usr/share/wordpress"

$hostname   = "app-server"

file {"site_link":
  path    => "$sitePath",
  ensure  => link,
  target  => "$siteSource",
  require => Package["wordpress"],
}

file {"wordpress_vhost":
  path    => "/etc/apache2/sites-enabled/000-default",
  ensure  => file,
  source  => template("wordpress-vhost.erb"),
  replace => true,
  require => [ Package["apache2"], Package["wordpress"], File["site_link"] ],
  notify  => Service["apache2"],
}

# Configuring the database access

$db-host = "db-server"

file {"wordpress_config":
```

```
    path    => "$configPath",
    ensure  => file,
    source  => template("wordpress-config.erb"),
    replace => true,
    require => [ Package["apache2"], Package["wordpress"] ],
  }

}
```

As we can see, PUPPET's code for the application server is pretty self-explanatory and clear. All the resources are grouped in a class `wordpress-server`, which signifies, that they should be instantiated exclusively on machines of this class (in our example we have of course only a single machine like that).

Probably the most interesting feature present in this piece of code are the declarations of relations between the resources, introduced using the attributes `require` (for a dependency relationship) and `notify` (for a refresh relationship). PUPPET will use this information to form a graph of resources and determine, in what sequence they should be treated.

Moreover, thanks to the refresh relationships, in the future PUPPET will react in cases when changes in one resource can affect another one. For example, if the configuration files for *PHP* or for the *Apache*'s Virtual Host are changed later on, PUPPET will automatically restart the `apache2` service, thus forcing it to reread these files and update its configuration.

It is also worth noticing, how all the keywords used in PUPPET's syntax underline its declarative character. We never provide an action to be performed on the resource (like "install this package"), we always provide a desired state of the resource (like "this package should be present").

```
# MySQL

class mysql-server {

  package {"mysql-server":
    ensure => present,
  }

  service {"mysql":
    ensure  => running,
    enable  => true,
    require => Package["mysql-server"],
  }

  $create-db-script-path = "/tmp/create-db.sql";

  file {"Create the Database Script":
    path   => $create-db-script-path,
    ensure => file,
    source => template("create-db.sql.erb"),
  }

  exec {"Create the Database":
    require     => [ Service["mysql"], File["Create Database Script"] ],
    command     => "mysql -uroot < ${create-db-script-path}",
    path        => "/bin:/usr/bin",
    unless      => "mysql -uroot -e \"use ${db-name}\"",
    refreshonly => true,
  }

}
```

Declarations in the class `mysql-server` seem to contain nothing surprising. The only aspect worth noticing here is use of the `exec` resource (which executes external shell commands) in order to create a database. As we cannot manage the database directly with none of the PUPPET's native resource types, we fall back on this pseudo type, which lets us incorporate execution of some external commands into the PUPPET's resource model in a quite elegant fashion.

3.6.2 Templates

Template file: wordpress-config.erb

```
/** The name of the database for WordPress */
define('DB_NAME', '<%= db-name %>');

/** MySQL database username */
define('DB_USER', '<%= db-user %>');

/** MySQL database password */
define('DB_PASSWORD', '<%= db-password %>');

/** MySQL hostname */
define('DB_HOST', '<%= db-host %>');
```

Template file: wordpress-vhost.erb

```
<VirtualHost *:80>
  ServerName <%= hostname %>
  DocumentRoot <%= sitePath %>

  <Directory <%= sitePath %>>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    allow from all
  </Directory>

</VirtualHost>
```

Template file: create-db.sql.erb

```
CREATE DATABASE <%= db-name %>;

GRANT ALL PRIVILEGES
ON <%= db-name %>*
TO '<%= db-user %>'
IDENTIFIED BY '<%= db-password %>';
```

PUPPET uses a standard *Ruby* template mechanism, which lets us embed any *Ruby* code in the template files. Here it is used simply to put the values of some variables (taken from our recipe execution context) in the right places of the configuration files.

3.6.3 Nodes classification

```
node app-server {
  include wordpress-server
}

node db-server {
```

```
  include mysql-server
}
```

If we imagine, that `app-server` and `db-server` are real DNS names of our two hosts, this piece of code simply defines, to which machine classes these two hosts belong, thus deciding which resources are relevant for each of them.

3.7 Summary

This is a synthetic summary of the PUPPET's characteristics. Full summary, including all the tools, can be found in the table 7 on page 41. Description of the employed terminology is available in the table 7 on page 42.

Model	domain	resources (e.g. file, package, service)
	relations	relations between resources: dependency, refresh
	control mechanism	declarations of desired state and behavior of resources
	hierarchy	flat model
Language	kind	own DSL
	style	declarative
	use	resources state
	expressivity	restrained
	pertinence	well adapted
Automatization	benefits	automatic converging of resources to the desired state
	configuration management	high
	orchestration	low
	requests	no
	solver	no
Architecture	architecture	centralized
	particularities	
Platforms	supported	all
	handling heterogeneity	medium

Table 3.1: PUPPET summary.

Chapter 4

Puppet with MCollective

MCollective is a framework to build server orchestration or parallel job execution systems. Basically: it is middleware (or rather meta-middleware). MCollective was bought by *Puppet Labs* and integrated into the Enterprise edition of Puppetin order to enhance it with some multi-server management and orchestration capabilities.

MCollective does not really change anything in Puppet itself. It merely creates an additional layer above the standard Puppet, which lets us control resources and machines on a slightly higher level.

4.1 Domain

Extending Puppet with MCollective does not really allow us to fully express and reference the localization of resources. We still cannot a define relationship between two resources which live on separate hosts, as each machine still operates in its own scope.

But there is something new: MCollective lets us discover all the resources of each machine and interact with them from the outside, from the perspective of the configuration server. Therefore from the external point of view we effectively obtain a unified model, which joins together two levels: machine level and infrastructure level. We can now see all the hosts and their respective resources on one big picture.

4.2 Automatization

We gain some degree of orchestration-level management. Thanks to MCollective, we can (manually or by writing client programs) manage whole groups of machines and interact with them: inspect their configuration, modify it and take actions based on it. Most importantly, this lets us prepare sequences of reconfigurations steps, containing quite complex orchestration logic, which can be easily executed on multiple hosts in a parallel fashion.

4.3 Summary

This is a synthetic summary of what MCollective adds to Puppet. Full summary, including all the tools, can be found in the table 7 on page 41. Description of the employed terminology is available in the table 7 on page 42.

		Puppet	what MCollective adds to Puppet
Model	domain	resources (e.g. file, package, service)	
	relations	relations between resources: dependency, refresh	
	control mechanism	declarations of desired state and behavior of resources	equivalent to sequences of desired states on all machines
	hierarchy	flat model	two levels: machines and their resources
Language	kind	own DSL	
	style	declarative	
	use	resources state	
	expressivity	restrained	
	pertinence	well adapted	
Automatization	benefits	automatic converging of resources to the desired state	infrastructure wide control of resources, orchestrating complex reconfigurations
	configuration management	high	
	orchestration	low	medium
	requests	no	
	solver	no	
Architecture	architecture	centralized	
	particularities		
Platforms	supported	all	
	handling heterogeneity	medium	

Table 4.1: Puppet with MCollective summary.

Chapter 5

Chef

Chef is a distributed configuration management tool created by *Opscode* and written in *Ruby*. It tries to achieve similar effects to Puppet with MCollective, but basing on slightly different principles: it abandons declarativity in favor of the imperative style. Some people say, Chef is supposed to be "Puppet done right". In reality it is rather "Puppet made imperative".

Particularities:

- Completely imperative (but it shows visible influences of the declarative approach).
- Dependency management is removed. It is replaced by the "order matters" philosophy.
- Has a lot of complex features and gives more direct power to the administrator. Unfortunately it makes Chef rather difficult to understand and master.
- Thin server – thick client approach.
- Some built-in service orchestration mechanisms.

5.1 Domain

5.1.1 What can we control?

In Chef, we have impression of describing resources, exactly like in Puppet. But because of the imperative nature of Chef, in reality we do not actually describe the *state* of resources, but rather *operations on* the resources: there is always an action involved. This imperative aspect is partially hidden by Chef's syntax, which may cause some confusion, as the things that we write look almost exactly like these in Puppet, but work differently.

Catalog of resources:

Most popular resources:

- File, Directory
- Package
- Service
- User, Group
- Execute

All the native resource types: Cron, Deploy, Directory, Env, Erlang Call, Execute, File, Git, Group, HTTP Request, Ifconfig, Link, Log, Mdadm, Mount, Ohai, Package, PowerShell Script, Remote Directory, Remote File, Route, Ruby Block, SCM, Script, Service, Subversion, Template, User

5.1.2 New types of Resources

In Chef, we can easily define new resource types by combining existing ones, just like in Puppet. Moreover, extending the resource model by writing plugins is not only possible, but quite sophisticated. Chef even features a dedicated simplified language for defining new types of resources with less effort, called LWRP (Lightweight Resources and Providers).

5.1.3 Relations

There are no dependency relationships in Chef. Only the order of resource declarations matters. Our lists of resource declarations are executed step by step similarly to an imperative program.

However, we have got a kind of imperative triggers to our disposition, which can be used to create links between actions. With their help, we can obtain effects similar to refresh relationships in Puppet.

5.1.4 Location

Resources exist only in a scope local to a single machine. The configuration description is distributed between many different types of objects and several layers of attributes, but at the end it is compiled to the final form on each concerned machine. Thanks to maintaining information about current state of all machines on the central server and because of introducing an advanced search feature, in Chef we obtain a global resource awareness level comparable to that of Puppet with MCollective.

5.2 Language

Chef uses a special extension of *Ruby* as its *Domain Specific Language*. It is completely imperative, but the syntax for describing resources is strongly mimicking declarativity. As an extension of *Ruby* it has of course the expressivity of a generic programming language.

5.3 Automatization

The way of doing things is Chef, which comes from the decision to avoid declarativity, results in a relatively decreased level of abstraction and modularity. However, this disadvantage is in some degree offset with gain of more control.

In Chef, configuration description is distributed between several different types of objects and a few layers of attributes. All that lets us define configuration in a very sophisticated and subtle ways, from coarse grain to fine grain. Moreover, Chef contains quite a few interesting ideas and refined mechanisms associated with handling and merging the configuration data dispersed among these various configuration objects.

Other significant Chef's features are the central configuration data store accessible by all the hosts (through objects called data bags) and an integrated advanced search mechanism. They do not only allow for *data driven* configuration and a certain level of *distributed discovery*, but also give Chef some interesting service orchestration capabilities.

5.4 Architecture

Chef features the standard centralized architecture with one server and multiple nodes. Configuration is distributed by the pull method.

One of Chef's principles is the "thin server – thick client" approach. Server acts only as a store of configuration data and most work is done by clients. This results in a high scalability.

5.5 Platforms

Chef supports all common platforms: *nix, Mac, Windows. Handling of heterogeneity is more or less the same level as in Puppet.

5.6 Example

5.6.1 Data bag

Data bag file: data_bags/wordpress-mysql/db-config.json

```
{
  "id"          : "db-config",
  "db-name"     : "wordpress-db",
  "db-user"     : "wordpress-db-user",
  "db-password" : "wordpress-db-password",
}
```

The data bag defined here serves the same purpose as common variables in the CFEngine3 and Puppet examples. In this particular case we cannot see any significant difference between these mechanisms, as both of them achieve the same result. But there is a difference: data bags are centrally stored configuration data, which can be accessed and modified by all the hosts. Thus with a data bag we can do some things, which are impossible to obtain using the solution based on simple common variables.

For example, instead of setting the configuration information centrally, like we do here, we could make our database server access the data bag and set this information by itself, after it has finished installing all the software components and creating the database. The application server would read it exactly as it is doing now. This way we would achieve a certain degree of true service orchestration, using the data bag as a communication channel between our two machines.

5.6.2 Cookbook wordpress

Recipe file: cookbooks/wordpress/recipes/default.rb

```
# Wordpress Server

# Apache2

package "apache2" do
  action :install
end

service "apache2" do
  action [:enable, :start]
end

# PHP5

package "php5" do
```

```
  action :install
end

package "libapache2-mod-php5" do
  action :install
end

# MySQL client + support

package "mysql-client" do
  action :install
end

package "php5-mysql" do
  action :install
end

# Wordpress
package "wordpress" do
  action :install
end

# Putting the site in place and adding a Virtual Host

siteName   = "mySite"
configPath = "/etc/wordpress/config-#{siteName}.php"

sitePath   = "/var/www/#{siteName}"
siteSource = "/usr/share/wordpress"

hostname   = "app-server"

link "site_link" do
  action      :create
  link_type   :symbolic
  target_file sitePath
  to          siteSource
end

template "wordpress_vhost" do
  action    :create
  path      "/etc/apache2/sites-enabled/000-default"
  source    "wordpress-vhost.erb"
  variables (
             :hostname => hostname,
             :sitePath => sitePath
            )
  notifies  :reload, "service[apache]"
end

# Configuring the database access

db-config = data_bag_item('wordpress-mysql', 'db-config')

db-host = "db-server"

template "wordpress_config" do
  action    :create
```

```
  path       configPath
  source     "wordpress-config.erb"
  variables (
             :db-name      => db-config['db-name'],
             :db-user      => db-config['db-user'],
             :db-password => db-config['db-password'],
             :db-host      => db-host
            )
end
```

This Chef recipe code is very easy to follow, because it has a form of an imperative program. In spite of looking quite similar to Puppet declarations, we do not really define resources here, but directly perform actions on them. Every step of this recipe will be executed exactly according to the given order.

The only exceptions to this rule are the notifications (the `notify` attribute). This is a solution used in Chef to provide a similar functionality as resource relationships in Puppet. But it is important to notice, that in case of Chef there is no abstract relationship between the resources, the notification mechanism is just a simple conditional trigger.

Here, for example, the action of creating an *Apache* Virtual Host configuration file from a template will trigger a restart of the `apache2` service. This will happen the first time this recipe is launched and then every time when the template file is changed (the action of creating the file from the template is skipped if the template file does not change, hence it does not launch the trigger).

Template file: cookbooks/wordpress/templates/default/wordpress-config.erb

```
/** The name of the database for WordPress */
define('DB_NAME', '<%= db-name %>');

/** MySQL database username */
define('DB_USER', '<%= db-user %>');

/** MySQL database password */
define('DB_PASSWORD', '<%= db-password %>');

/** MySQL hostname */
define('DB_HOST', '<%= db-host %>');
```

Template file: cookbooks/wordpress/templates/default/wordpress-vhost.erb

```
<VirtualHost *:80>
  ServerName <%= hostname %>
  DocumentRoot <%= sitePath %>

  <Directory <%= sitePath %>>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    allow from all
  </Directory>

</VirtualHost>
```

Templates in Chef use the same mechanism, as these in Puppet. There is just one little difference in the template evaluation process: the variable values used in the template are not taken directly from the recipe's execution context, but they have to be specified explicitly in the action's description, as a list of keys with values.

5.6.3 Cookbook mysql

Recipe file: cookbooks/mysql/recipes/default.rb

```
package "mysql-server" do
  action :install
end

service "mysql" do
  action [:enable, :start]
end

create-db-script-path = "/tmp/create-db.sql";

db-config = data_bag_item('wordpress-mysql', 'db-config')

template "Create the Database Script" do
  action    :create
  path      create-db-script-path
  source    "create-db.sql.erb"
  variables (
             :db-name     => db-config['db-name'],
             :db-user     => db-config['db-user'],
             :db-password => db-config['db-password']
            )
end

execute "Create the Database" do
  command "mysql -uroot < #{create-db-script-path}"
  path    "/bin:/usr/bin"
  not_if  "mysql -uroot -e \"use ${db-name}\""
end
```

Template file: cookbooks/mysql/templates/default/create-db.sql.erb

```
CREATE DATABASE <%= db-name %>;

GRANT ALL PRIVILEGES
ON <%= db-name %>*
TO '<%= db-user %>'
IDENTIFIED BY '<%= db-password %>';
```

As we can see, the recipe for our database server and the template linked with it are not kept together with the previously presented files, but are put in a separate cookbook. This way we obtain a better separation of code and configuration data linked with two different machines. This separation is a little easier to achieve in Chef, than in Puppet, as thanks to the data bags mechanism we do not need to have common variables declared in both recipes.

5.6.4 Roles

Role file: roles/app-server.rb

```
name "app-server"
run_list [ "recipe[wordpress]" ]
```

Role file: roles/db-server.db

```
name "db-server"
run_list [ "recipe[mysql]" ]
```

This code defines two roles (`app-server` and `db-server`), precising which recipes should be run on machines belonging to these roles. We do not see however, how these roles are assigned to particular machines; this is done in the respective nodes' configuration.

5.7 Summary

This is a synthetic summary of the Chef's characteristics. Full summary, including all the tools, can be found in the table 7 on page 41. Description of the employed terminology is available in the table 7 on page 42.

Model	domain	resources (e.g. file, package, service)
	relations	imperative triggers between actions on resources
	control mechanism	declarations of actions on resources
	hierarchy	flat model
Language	kind	an extension of *Ruby*
	style	imperative
	use	actions on resources
	expressivity	full
	pertinence	well adapted
Automatization	benefits	fine grain control of distributed configuration, some orchestration capabilities
	configuration management	very high
	orchestration	medium
	requests	no
	solver	no
Architecture	architecture	centralized
	particularities	thin server, thick client
Platforms	supported	all
	handling heterogeneity	medium

Table 5.1: Chef summary.

Chapter 6

juju

JUJU is *Canonical*'s tool for deploying and orchestrating services in the cloud. It is still in a quite early phase of development.

Particularities:

- Focused on a higher level of abstraction in configuration's control: we can say, that JUJU manages "services in a cloud", not "resources on machines".
- Introduces direct relations between "deployed services".
- But it has two big downfalls:
  - Management of configuration of each single machine is quite primitive.
  - The model of "deployed services" is very restricted.
- Works only with *Ubuntu*.

6.1 Domain

6.1.1 What can we control?

In, JUJU we do not control resources on machines, we manage entities of a higher level, called *services*. Here, a service is a (possibly distributed) application, integrated into the JUJU framework as an individual component. Another crucial concept is a relation: a link between services, an orchestration mechanism permitting them to coordinate their configuration and work together.

A deployed service has a form of one or many uniform service units, each occupying one virtual machine in our cloud environment. Service unit is a running instance of the service. We can dynamically scale a service up or down simply by adding or deleting its service units.

In order to define services and their possible relations, we write charms. A charm is no more than a set of low-level configuration scripts and some metadata. It contains all the necessary information and code to let JUJU automatically deploy and manage service units of the given service.

Unfortunately, the high level perspective and broad strokes philosophy of JUJU has a downside: the mechanisms responsible for controlling the configuration of each single machine are very primitive and there is no resource model whatsoever. Everything is based on bare configuration scripts, integrated into charms as so-called hooks and automatically launched on various events.

6.1.2 Relations

The only type of relationships between software components in JUJU are service-to-service relations. A relation can be established only between two services with compatible interfaces:

- one service must require a certain type of relation,
- the other service must provide it.

Relations are defined and controlled on the abstract level of whole services, but are implemented and realized on the level of service units, as true machine-to-machine connections. For example, when we order our web application service to connect to our database service, in effect each of our application servers (if it is a service unit of this particular web application service) will establish a connection with the database server.

6.1.3 Location

Every service deployed in our cloud environment is a separate entity. We can have several services of the same type (i.e. based on the same charm) deployed at a time, each with a unique identity.

On the other hand, service units forming together a single service are all homogeneous and unrecognizable. We should be able to add or delete them at will, without thinking, which particular instance is affected.

6.2 Language

JUJU does not feature any dedicated domain specific language. Any executable file can be used as a script for a charm (but typically they are just bash scripts). Metadata for charms is written in specific YAML formats.

6.3 Automatization

JUJU gives us a high-level service model and lets us very easily deploy services, bind them or unbind them and scale them up or down at will. However, in order to achieve this simplicity, it has to restrain itself greatly. For example:

- it permits only a very rigid service structure: a service can contain only multiple completely homogeneous service units,
- it allows us to deploy only one service unit per machine,
- it does not provide any real means to correctly handle the updates of software packages.

In JUJU we can spot quite a few interesting solutions, like the mechanism responsible for coordination of distributed agents and enforcing the infrastructure configuration modifications, based on *Apache*'s *ZooKeeper*.

Distributed configuration aspect of JUJU is not well developed. There is no machine-level resource model nor machine-level resource management. The configuration management performed on hosts is based mostly on bare configuration scripts, hence it is very primitive. On the other hand, the service orchestration aspect of JUJU is obviously more developed and direct than in any of the other tools which we have seen.

6.4 Architecture

JUJU features a centralized architecture: a single central *bootstrap node* (the first virtual machine brought up by JUJU when setting up the cloud environment) is responsible for creating, deleting and coordinating all the other nodes. Configuration is distributed by the push method.

6.5 Platforms

According to its developers, JUJU eventually will be capable to work with different cloud environments. For now it supports only *Amazon EC3*. JUJU is (by design) compatible with only one platform: *Ubuntu Cloud OS*.

6.6 Example

6.6.1 Charm wordpress

Metadata file: metadata.yaml

```
name: wordpress
revision: 1
requires:
  db:
    interface: mysql
```

This is a JUJU metadata file for the service `wordpress`. Our `wordpress` service can be a client in a single `mysql` relation with another service (hence the `requires` keyword, the matching server side service needs to have a `provides` declaration for a similar interface type). Single service can potentially engage in multiple different relations, `db` is an internal identifier of this particular relation.

Hook file: install

```
#!/bin/bash

# Installing packages.
apt-get -y install apache2 php5 libapache2-mod-php5 mysql-client php5-mysql

# Putting the site in place and adding a Virtual Host.
sitePath="/var/www/mySite"
siteSource="/usr/share/wordpress"

# Adding the symbolic link.
ln -s $siteSource $sitePath

# Preparing the Virtual Host.
hostname=`unit-get private-address`

virtualHostTemplate="""
<VirtualHost *:80>
  ServerName $hostname
  DocumentRoot $sitePath

  <Directory $sitePath >
    Options Indexes FollowSymLinks MultiViews
    AllowOverride None
    Order allow,deny
    allow from all
  </Directory>

</VirtualHost>
"""

# Setting up the Virtual Host.
```

```
echo "$virtualHostTemplate" > /etc/apache2/sites-enabled/000-default

# Restarting Apache
/etc/init.d/apache2 restart
```

As we can see, the `install` hook file is just a bash script (but it could be any type of an executable file), which installs all the required software components on the host. It will be automatically executed when our `wordpress` charm is being deployed on a new virtual machine.

Hook file: db-relation-changed

```
#!/bin/bash

# Getting the database configuration
db_name=`relation-get database`
db_user=`relation-get user`
db_password=`relation-get password`
db_host=`relation-get hostname`

# All values are set together, so checking
# on a single value is enough.
# If $user is not set, database is still setting
# itself up, so we exit awaiting next run.
[ -z "$user" ] && exit 0

# Configuring Wordpress to work with the database
configPath="/etc/wordpress/config-mySite.php"

configTemplate="""
/** The name of the database for Wordpress */
define('DB_NAME', '$db_name');

/** MySQL database username */
define('DB_USER', '$db_user');

/** MySQL database password */
define('DB_PASSWORD', '$db_password');

/** MySQL hostname */
define('DB_HOST', '$db_host');
"""

echo "$configTemplate" > $configPath

# Opening the port 80
open-port 80/tcp
```

In the `db-relation-changed` hook we can see, how the service orchestration works on the low level in JUJU. The `db` relation is used here as a communication channel: on the remote side the database server sets some parameters and on this side we can read them using the `relation-get` command. The hook's filename itself determines, which relation is concerned. This hook will get (re)executed every time, when a remote host joins the `db` relation or changes one of its parameters.

6.6.2 Charm mysql

Metadata file: metadata.yaml

```
name: mysql
revision: 1
```

```
provides:
  db:
    interface: mysql
```

The metadata file of the `mysql` charm declares a relation exactly symmetrical to the one from the `wordpress` charm.

Hook file: install

```
#!/bin/bash
apt-get -y install mysql-server
service mysql restart
```

Hook file: start

```
#!/bin/bash
service mysql start
```

Hook file: stop

```
#!/bin/bash
service mysql stop
```

Hook file: db-relation-changed

```
#!/bin/bash

# Getting the remote unit's service name.
$serviceName=`echo $JUJU_REMOTE_UNIT | cut -d '/' -f 1`

# Preparing the database config.
$dbName="$serviceName"
$dbUser="dbUser"
$dbPassword="dbPassword"

# Creating the database if it doesn't exist.
if $(mysql -uroot -e "use $dbName"); then
    # Database already exists - do nothing.

else
    # Database doesn't exist - create it.

    # Preparing the SQL code to create the database.
    $createDatabaseScriptTemplate="""
CREATE DATABASE $dbName;

GRANT ALL PRIVILEGES
ON $dbName*
TO '$dbUser'
IDENTIFIED BY 'dbPassword';
"""

    # Creating the database.
    mysql -uroot -e "$ceateDatabaseScriptTemplate"
fi

# Setting the relation db settings.
```

```
hostname=`unit-get private-address`

relation-set database=$dbName
relation-set user=$dbUser
relation-set password=$dbPassword
relation-set hostname=$hostname
```

And here we see, what happens on the other side of the `mysql` relation, which we have already observed from the `wordpress` service's perspective. When a remote host (application server) joins the relation, the database server will create a new database for him (if it does not already exist) and will set all the relation parameters accordingly, using the `relation-set` command (symmetrical to `relation-get`).

6.7 Summary

This is a synthetic summary of the JUJU's characteristics. Full summary, including all the tools, can be found in the table 7 on page 41. Description of the employed terminology is available in the table 7 on page 42.

Model	domain	services (e.g. mysql, wordpress)
	relations	relations between services: provides, requires
	control mechanism	definitions of services and their possible relations
	hierarchy	flat model
Language	kind	YAML for metadata / any executable language for scripts
	style	declarative / imperative
	use	services' metadata / configuration scripts for deploying and managing services
	expressivity	restrained / full
	pertinence	well adapted / too generic and low-level
Automatization	benefits	deployment and orchestration of services, only very primitive resource management
	configuration management	low
	orchestration	high
	requests	no
	solver	no
Architecture	architecture	centralized
	particularities	cloud only
Platforms	supported	Ubuntu Linux
	handling heterogeneity	none

Table 6.1: JUJU summary.

Chapter 7

Conclusion

We have analyzed in detail a comprehensive selection of tools that are widely used for *distributed configuration management*, by allowing to define, store and maintain reusable configuration artifacts, and offering means to automate the process of enforcing configuration changes on multiple hosts.

For each of these tools, that are quite different in their philosophy and implementation, we have identified several distinguishing characteristics that we consider essential to establish a profile useful to compare them.

These characteristics fall into the following five main categories.

Domain
The kind of entities that the tool can manage explicitly, as well as how they are modeled and structured.

Language
The kind of languages available for the user to describe a configuration or the requirements on a configuration.

Automation
The mechanisms offered by the tools to automate configuration management, and to compute (optimal) reconfiguration sequences.

Architecture
The structure and organization of the tool.

Platforms
The kind of operating systems supported, as well as the kind of support for heterogeneity.

The summary of these profiles is collected in the table 7 on the next page, that can be used to quickly compare them. The terminology employed in this table is explained in the lexicon showed in table 7 on page 42.

	CFEngine3	Puppet	what MCollective adds to Puppet	Chef	juju
Model					
domain	resources (e.g. file, package, service)	resources (e.g. file, package, service)		resources (e.g. file, package, service)	services (e.g. mysql, wordpress)
relations	none	relations between resources: dependency, refresh		imperative triggers between actions on resources	relations between services: provides, requires
control mechanism	promises about resources' state and behavior	declarations of desired state and behavior of resources	equivalent to sequences of desired states on all machines	declarations of actions on resources	definitions of services and their possible relations
hierarchy	flat model	flat model	two levels: machines and their resources	flat model	flat model
Language					
kind	own DSL	own DSL		an extension of *Ruby*	YAML / any executable language
style	declarative	declarative		imperative	declarative / imperative
use	actions on resources	resources state		actions on resources	services metadata / configuration scripts for deploying and managing services
expressivity	restrained (but quite high)	restrained		full	restrained / full
pertinence	cumbersome, too declarative	well adapted		well adapted	well-adapted / too generic and low-level
Automatization					
benefits	equivalent to configuration scripts execution framework	automatic converging of resources to the desired state	infrastructure wide control of resources, orchestrating complex reconfigurations	fine grain control of distributed configuration, some orchestration capabilities	deployment and orchestration of services, only very primitive resource management
configuration management	high	high		very high	low
orchestration	low	low	medium	medium	high
requests	no	no		no	no
solver	no	no		no	no
Architecture					
architecture	distributed	centralized		centralized	centralized
particularities	by default simulates centralized architecture			thin server, thick client	cloud only
Platforms					
supported	all	all		all	Ubuntu Linux
handling heterogeneity	low	medium		medium	none

Table 7.1: The comparison summary table.

Model

domain: What are the configuration objects, which we manipulate?

relations: What relations can we define in the domain?

control mechanism: In what way do we control the objects of the domain? What form do our decisions about the objects' properties take?

hierarchy: Are the resources organized in some kind of a hierarchy? A *hierarchy* means here, that some resources can be embedded into others. In a flat model there is no such possibility.

Language

(Note: JUJU uses two distinct language types for two different aims.)

kind: What kind of programming language is used in the tool? (DSL stands for Domain Specific Language)

style: What programming paradigm does the language primarily support?

use: How is the language used to interact with resources? More precisely: what do we write using the language?

expressivity: How much is the language restrained? Is it completely domain specific or rather close to a generic programming language?

pertinence: Is the level of abstraction of the language well adapted to its task? Or is the language too restricted or too generic?

Automatization

benefits: What are the benefits of using the tool over doing things "by hand"? What does the tool give us?

configuration management: How well is the tool adapted in practice to handling the distributed configuration management?

orchestration: What level of capabilities in orchestrating complex multi-host reconfigurations does the tool possess?

requests: Does the tool support high level reconfiguration requests?

solver: Does the tool employ a solver / planner of some kind when preparing a reconfiguration, in order to find optimized reconfiguration solutions / plans?

Architecture

architecture: What is the tool's architecture (and hence the structure of the environment governed by it)? Does it take the form of a central server and a pool of managed hosts (centralized architecture)? Or can it support different structure forms (distributed architecture)?

particularities: Does the tool's architecture have any particular characteristics?

Platforms

supported: Which platforms can be managed using the tool? ("all" means here: *NIX, Mac, and Windows operating systems)

handling heterogeneity: How high is the tool's capability of abstracting over differences between platforms? Do we access resources on different platforms using a uniform interface?

Table 7.2: The summary legend.

7.1 Summary of capabilities and limitations

By simply looking at the summary table, it is easy to see that all of these tools miss a high level automation layer: it is not possible to specify high level requests, and have the tool rely on an external, efficient constraint solver or planner to propose a reconfiguration plan, let alone an optimal one; the best that one can find is Puppet's architecture that is the only one built on top of a clear notion of *resource*, and that provides mechanisms whose goal is to *converge* a system towards a particular configuration.

We also notice, looking at the *relations* line, that none of these tools allows to declare incompatibilities among resources, while conflict relations do exist among the underlying components: this is a serious limitation of all the models, which are not faithful, and may lead to surprising and inconsistent configurations.

We also remark that none of the tools has a real global model of resources, which allows to declare relationships between resources from different machines: either we find a flat model, describing only resources on a single machine, like in CFEngine3, Puppet, and Chef, which are the most used ones; or we find a flat model describing only services in the cloud, like juju in which each service is identified to a different machines; MCollective does add a notion of collection of machines on top of Puppet, but without a unified model.
This is a serious limitation, as it is currently impossible to declare dependencies between services, software components and machines in a unified way, while this is an essential precondition for being able to express sophisticated reconfiguration requests.

Finally, we notice that all these tools only allow to talk about a single configuration of a system, be it the one deployed now, or the one that should be deployed next, so there is no explicit notion of reconfiguration plan.

7.2 Relationship with the Aeolus project

It appears now clear that none of these tools is able to perform the tasks of reconfiguration planning which are at the core of the Aeolus project, but some of them can be seen as potential target deployment layers for reconfiguration plans produced by the solvers developed in the framework of the Aeolus project. We believe in particular that any of CFEngine3, Puppet, and Chef, could serve this role.

Bibliography

[1] The *CFEngine* company homepage, http://cfengine.com/, 2012.

[2] The *Puppet Labs* company homepage, http://puppetlabs.com/, 2012.

[3] The *Opscode* company homepage, http://www.opscode.com/, 2012.

[4] The *juju* project homepage, https://juju.ubuntu.com/, 2012.

Appendix A

Tool details

In this appendix you can find more details about the presented technologies. They are not necessary to understand the key aspects of the comparison we have performed in the main part of this document, but might be useful to evaluate the applicability of the technology, in the context of Aeolus or elsewhere.

A.1 CFEngine3

CFEngine3 is one of the configuration management tools with the longest history (first version dates back to 1993). It has strongly influenced other configuration management software, like *Puppet* and *Chef*. CFEngine3 is based on the *Promise Theory*, developed by its creator Mark Burgess.

A.1.1 Idea

All the hosts in the environment document their intentions to act or behave in a specific way by making promises.

Normally (without promises), each host is completely independent and autonomous in his behavior: it cannot be influenced from outside and compelled to do something. But if we want to control its configuration, we need to make it cooperate with other hosts and depend on them. In order to achieve this kind of dependence, we use promises to enforce a certain hierarchy of configuration definition and propagation.

In the most simple case, we create a single policy distribution server (where all the configuration decisions are stored) and force every host to make a promise, that he will download the configuration information from the server and apply it to himself in constant intervals. As the configuration description also takes a form of a list of promises, we can say that each of our hosts simply synchronizes his promises with the central policy distribution server.

This schema represents the typical architecture of a distributed configuration management tool: one central configuration server and multiple configuration clients, who automatically conform to the configuration defined for them on the server. CFEngine3 by default follows that schema, but it is not bound by it. In CFEngine3 every host contains both a client and a server component, and the role we assign to it in our infrastructure is decided completely by us. We control not only the distributed configuration itself, but also the way *how* it is distributed. The standard scaffolding of promises implements the habitual architecture and behavior presented above, but it is not obligatory.

For example, we could design a more robust system based on peer-to-peer model, where hosts exchange configuration information directly between themselves, without need to connect every time with the central server. And if we want to, in CFEngine3 we are able to modify the standard promises scaffolding in order to implement this architecture.

A.1.2 Policy and Promises

A configuration policy is a set of all the decisions concerning both the configuration of our whole environment and the methods employed to propagate that configuration between all the hosts. A promise is a fundamental statement in CFEngine3, we can regard it as a policy atom.

Each promise is composed of three parts:

- The promiser: the object that formally makes the promise.
  (Usually we interpret this concept in a slightly different way: it is the host who makes a promise about this object's state or behavior.)
- The promisee (optional) : Someone interested in the outcome of the promise.
  (Used purely for documentation and knowledge management.)
- The body : Contract of the promise, defining what state and behavior the promiser object obliges itself to retain.

There are many types of promises available in CFEngine3. Most of them have to be directly associated with a specific concrete object on the host, like a file or a software package (which is their promiser), and they are supposed to manage that object's state or behavior according to the contract we specify (their body).

We have also some promise types used only internally in CFEngine3, which do not affect directly any object outside of it: their promisers are ephemeral internal CFEngine3 objects (or pseudo-objects), serving various aims. These CFEngine3's special internal promise types are in fact a way to introduce some functionality to the promise declarative language instead of adding more language primitives.

Normal Promises

The type of the promise obviously imposes which kind of objects can be used as its promiser (e.g. the promiser of a file promise needs to be a certain file in the filesystem) and which kind of contracts we can specify in its body (e.g. in a file promise we can specify this file's content, ownership, permissions, etc).

These are the most popular promise types, which let us manage real objects existing on the host:

- file: files and directories in the filesystem.
- package: software packages.
- command: arbitrary shell commands.
- process: running programs.
- service: a set of processes and commands bundled together.
- interface: network interfaces.
- storage: disks and filesystems.
- database: tables in databases (SQL, LDAP, MS Registry)
- environment: enclosed computing environments (physical machines, virtual machines, clouds)

Internal Promises

These are two important promise types, which are used internally in CFEngine3:

- variable: definitions of variables.
- class: definitions of conditions.

A detailed explanation of how these types of promises work can be found in sections A.1.3 and A.1.3

A.1.3 Language

We define promises using the CFEngine3 declarative syntax.

Syntax

```
promises_definitions =
    type   , ':' ,
  [ class  , '::' ] ,
  { promise_definition  , ';' };

type  = ? available promise types ? ;
class = ? available class ? | expression returning a class ;
```

```
promise_definition =
    promiser , [promisees] ,
    attribute , '=>' , value ,
  { ',' , attribute , '=>' , value } ;

promiser  = string ;
attribute = ? attributes available for this promise type ? ;

promisees =
  '->' , '{' , promisee , { ',' , promisee } , '}' ;
```

Example

```
files:
   linux::
      "/tmp/promiser"
         create => "true";
```

In this example we declare a promise of the type `file`, which belongs to the class `linux`. Its promiser is the file with path `/tmp/promiser`. Its body contains one attribute `create` with value `true`.

We can read this code as:

> I promise, that if I am a Linux machine I will create a file at the path `/tmp/promiser`.

As we can see, this promise is quite vague: neither the content nor the permissions of the newly created file are precised.

Bundles

A BUNDLE is a container grouping together multiple promises. In the CFENGINE3's declarative language it serves a purpose similar to a sub-routine: allows modularity and reusability of the code.

Syntax

```
bundle =
   'bundle' , bundle_type , bundle_identifier ,
   '{' , { promises_definitions } , '}'
```

Example

```
bundle agent mybundle {

files:
   "/etc/ssh/sshd_config"
       copy_from => secure_cp("/var/cfengine/masterfiles/sshd_config",
                              "serverhost");

packages:
   "openssh-server"
      package_policy => "add",
      package_method => apt;

services:
   "sshd"
```

```
      service_policy => "start";

}
```

In this example we declare a bundle of three different promises: one about a file, one about a software package and one about a service. We can read their code as:

1. I promise to create a file at the path `/etc/ssh/sshd_config` and copy its content from a remote file taken from the server.
2. I promise to install a package `openssh-server` using the package manager `apt`.
3. I promise to start a service `sshd`.

As these promises are in one bundle, they should be always executed together.

Body Attachments

The body of each promise simply contains a list of attributes with values. But sometimes, in order to describe well certain more complex aspects of configuration, we need more complicated data structures (generally: a tree of nested key-value pairs).

CFEngine3 handles this problem using a mechanism of BODY ATTACHMENTS. Instead of directly providing a value for an attribute, we can put a pointer to a parameterizable list of attributes, which is defined somewhere else as a body attachment. This way we are plugging the list, represented by the body attachment, in the given place, and we are effectively creating a tree structure.

Note: We are not allowed to put the whole tree structure inline, we always have to use the body attachments mechanism. Therefore we always operate on lists with pointers to other lists, we never construct the whole trees in our code.

Syntax

Declaration of the body attachment:

```
body_attachment =
   attribute , body_identifier ,
   '(' , [ parameter , { ',' , parameter } ] , ')' ,
   '{' , { attribute => value } , '}' ;
```

Using the body attachment declared before:

```
type:
  class::
    promiser
      attribute_1 => value_1 ,
      attribute_2 => value_2 ,
      ...
      attribute_i =>
         body_identifier ( parameter_1_value, parameter_2_value, ... ) ,
      ...
      attribute_n => value_n
```

Note: `attribute_i` in the promise has to match the `attribute` in the body attachment declaration. We can see it as a way to precise the type of the body attachment: it can be used in any number of promises, but can be linked only with exactly this kind of attribute.

Example

```
files:
   "/tmp/promiser"
      create => "true",
      perms  => myperms(644);
```

```
body perms myperms(mode) {
   mode   => "$(mode)",
   owners => { "me", "you" },
   groups => { "users", "sysadmins" };
}
```

In this example we declare a body attachment for attribute `perms` (it means, that this body attachment can be only used with promise attributes called `perms`), with an identifier `myperms` and one parameter: `mode`. It has three attributes. The value of the attribute `mode` is using directly the parameter `mode`.

This body attachment is instantiated in the declaration of a sample promise as the value of its attribute `perms`. The identifier `myperms` is used to reference it and `644` is passed as the parameter `mode`.

We can imagine, that it creates a tree of key-value pairs which looks like that:

```
body of the Promise
 ├─ create => "true"
 └─ perms
    ├─ mode => "644"
    ├─ owners => { "me", "you" }
    └─ groups => { "users", "sysadmins" }
```

Variables

CFEngine3 has a peculiar way to introduce variables to the promise declarative syntax: we declare and assign variables using special promises of the type "variable".

It means, that we cannot directly assign value V to the variable X using the classic assignment mechanism. Instead we are obliged to declare a promise, that the variable X will have a value equal V.

Example

```
vars:
  "x" string => "aaa";
  "y" int    => "42";
```

It is worth noticing, that in this example both `string` and `int` are simply attributes of their corresponding promises. So we should read it as:

> Variable X promises, that its string value will be equal "aaa"
> and
> variable Y promises, that its integer value will be equal 42.

Classes

CFEngine3 has also an unusual way of introducing conditions into its language. We call conditions "classes" and we can test them before any promise: if the condition embodied by a class is true, the promise gets executed, if not – the promise is not taken into account.

This mechanism seems well adapted to certain cases, i.e. for conditions which can be naturally mapped to certain classes of machines (like database servers, application servers, proxy servers,

etc). In this context it becomes obvious, what putting a class "database-server" before a given promise means: this promise belongs to the class "database-server", it is relevant only for database servers and should be ignored everywhere else.

However, this approach becomes quite unnatural if we want to express most of the other types of conditions. For instance, a condition "does a file `/etc/passwd` exist?" or "are we Tuesday?" does not form a natural class of machines.

There are two general categories of classes. CFEngine3 discovers by default some information about its host and about the local environment and puts it in so-called *hard classes*. They let us test some basic conditions about the operating system's architecture, the unqualified name of the host and time-related facts like current date, hour, weekday etc.

We can also define our own *soft classes* by expressing them as promises of the type "class". In the definitions we can use some operators and built-in functions prepared for manipulating classes. For example:

```
"solarisOrLinux"   expression => "linux||solaris";
"passwdFileExists" expression => fileexists("/etc/passwd");
```

In this piece of code we declare:

- a class `solarisOrLinux`, which embodies a condition true only if we are on a host running *Linux* or *Solaris*
- a class `passwdFileExists`, which embodies a condition true only if the file `/etc/passwd` exists on the host.

Distributed discovery

CFEngine3 supports a concept called *distributed discovery*. This means, that it allows us to design our configuration policy in a way to make hosts partially self-reliant. A host can detect other hosts in the environment and sample their behavior; then it can base some parts of its own configuration on the data acquired this way. For example, a load balancer can check periodically, which application servers are currently running, and change his internal settings accordingly.

In theory, the *distributed discovery* is a great and very useful idea, which lets us introduce some between-hosts relationships and dependencies in our environment. Unfortunately, in CFEngine3 the *distributed discovery* capabilities are implemented on rather low level of abstraction. Essentially this feature is implemented as just a little addition to the CFEngine3 language: a number of special built-in environment-probing functions. These special functions, while useful, are usually quite low-level and not capable of interacting with the promise layer on other machines. Therefore they do not give us a possibility of creating complicated and interesting constructs, like real promise-to-promise relationships between hosts. But we can still define the configuration of one machine with respect to other machines in some degree.

Example of an environment-probing function

```
selectservers(
   [list of hosts to probe],
   [port number],
   [query string],
   [regular expression to match successful response],
   [maximum number of bytes to read],
   [name of array for results]
)
```

This is a function that probes a selected list of hosts and filters only these, which pass a certain test.

It simply sends a fixed `query string` to the given `port number` of each host from the `list of hosts to probe` and awaits answer (reading up to the `maximum number of bytes`). If the

answer matches the given `regular expression`, then the host passes the test and is included in the `array for results`.

It is worth noticing, that the list of hosts has to be known beforehand, so there is not a lot of actual *discovery* included in this process.

Distributed discovery example

```
vars:
   "hosts"       slist => {"host1", "host2", ... };
   "up_servers" int   => selectservers{"@hosts", "80", ... , "alive_servers"};

classes:
   "someone_alive" expression => isgreaterthan("$(up_servers)", "0");
   "i_am_a_server" expression => regarray("up_servers", $(fqhost));

reports:
   someone_alive::
      "Number of active servers is $(up_servers)" action => always;
```

This example illustrates, how we can use the CFEngine3 *distributed discovery* capabilities:

A given list of hosts ({"host1", "host2", ...}) is probed using the `selectservers()` function. Only the hosts which are up and running (i.e. respond correctly to some kind of request sent to the port 80) are selected and put in the list `up_servers`.

Then two potentially useful conditions based on that list are defined as *soft classes*:

- condition `someone_alive` is true when at least one host is up
- condition `i_am_a_server` is true only if our host is one of the running servers.

After that, if at least one server is running (i.e. if the condition `someone_alive` is fulfilled), a report containing the number of currently active servers is generated.

Dependencies and the normal ordering

Promises are always executed on a bundle-by-bundle basis. This means that all the promises from one bundle are treated before passing to the next bundle.

The sequence of executing promises in each single bundle is also determined: they are executed according to so-called NORMAL ORDERING. This ordering is basically a fixed sequence based on a common logic of how things should work in most cases. It means for example, that we treat all the files before the software packages and all the packages before the services. If there are two actions for one file – creation and deletion – we always first delete, then create the file and not the other way around. And so on.

This mechanism is in fact the main method offered by CFEngine3 to manage dependencies between the promises within a single bundle. There are several other mechanisms, but none of them is really complete (for various reasons).

We can force a dependency between two promises using an indirect technique, involving two more elements: by making a promise in a class context and defining that class based on the outcome of another promise.

We can also document a dependency using one of two simple mechanisms:

- promisees : By putting one promise's name in the list of promisees of a second promise, we indicate, that first of them affects the second one (so the second one depends on the first one).
- `depends_on` attribute : We can add a `depends_on` attribute to the body of a promise and put a second promise's name as its value. This indicates, that the first one depends on the second one.

Unfortunately both the promisees and the `depends_on` attribute are there only for the sake of knowledge management. Their sole purpose is to let the creators and maintainers of the configuration policy document dependencies between components when writing promises. This information is only used for their own knowledge, as CFEngine3 does not perform any automatic reasoning using it. (Although apparently in the commercial version of CFEngine3 some of this data is actually used to "track the dependencies between [...] promises and map out impact analyses".)

However, there is one type of promises which natively supports dependencies: services. The body of a service promise can contain a special attribute `service_dependencies`, to define a list of services that must be already running before this service can be started.

Particularity of the language

The CFEngine3's declarative language, which we use to define promises, has certain advantages, but also a few drawbacks. On the one hand it is really completely declarative, in the same time remaining rather powerful and expressive. On the other hand however, it often appears to be very cumbersome and unnatural, even when we want to express quite straightforward concepts.

That means more or less, that although we can usually succeed to express what we want with the CFEngine3 declarative language (i.e. to describe the configuration policy which we wish to apply), sometimes even doing simple things can entail a lot of trouble (i.e. writing many lines of carefully planned code just to get around some obstacles imposed by the language itself).

For example, most known algorithms make heavy use of two basic operations: assigning a value to a variable and testing a condition. In CFEngine3's language, the indirect mechanism of introducing variables and conditions as promises, combined with lack of simple means to enforce a dependency between two promises (and thus explicit some kind of temporary order of actions) makes implementing many simple algorithms unusually complicated.

Of course we can interpret it as a typical problem of the programming paradigm: the language is a tool and it's designed to solve efficiently only certain type of problems. In fact usually we should not attempt to write explicit algorithms using a declarative language. However, in this case we find a certain contradiction in semantics of the promise description language (more details about how promises are executed are in paragraph A.1.4 on the next page). The language itself is highly declarative, but the nature of objects which we declare is imperative. In fact we declare actions. So it is only natural, that sometimes we want to define the order of execution of these actions (and want to program some algorithms).

A.1.4 Framework

CFEngine3's specificity lies in the fact, that it contains very few built-in behaviors and constraints. In other words: it does not make absolute decisions neither about the configuration itself, nor about the methods, which are employed to propagate the configuration between the hosts in our environment. All the aspects of its behavior are implications of the defined configuration policy and therefore can be changed. (CFEngine3 does not actually do too much just by itself: without declaring any promises it does not do anything at all.)

This design decision has two important repercussions:

- On the one hand, CFEngine3 is very flexible and adaptable to many different infrastructures and various applications (e.g. as it does not need a reliable network nor a reliable infrastructure to work, it can be used in circumstances where many other configuration management tools would fail). It lets us write our own configuration distribution and management mechanisms using only promises.

- On the other hand, in most cases when we need to manage configuration in a distributed system, all we want is a standard, centralized "one server – many hosts" architecture, where all hosts simply synchronize their configuration with the server from time to time. And in CFEngine3 in order to achieve that behavior, we already need a certain scaffolding of basic promises. There is no built-in mechanism of automatic configuration propagation. Each

host has to promise explicitly to download his configuration from the single host, which is playing the role of the server.
This problem is partially outbalanced by the fact, that this standard scaffolding is directly given to us by CFEngine3: it is implemented by the default promises present on each host. But it does not change the conceptual inconvenience, as the very presence of this bottom layer in CFEngine3 violates the abstraction boundaries of software components configuration management.

Summarizing: all that results in a tool that is powerful and flexible indeed, but which unfortunately does not let abstract completely from the low-level aspect of the configuration control.

The agent's run

Agent is the part of CFEngine3 responsible for executing promises on a host. It is usually activated at constant intervals of time, but on a special demand it can also be forced to run out of normal schedule.

When an agent is run, it does one main thing: it reads and executes the main promises file (by default stored at the path `inputs/promises.cf`). All the other files to include and promises to execute have to be mentioned there.

The main promises file

The key part of the main promises file is the common control body. It lets us specify two settings:

- The list of additional promise files which need to be read.
- The sequence of bundles which have to be executed.

This is an extract of the default main promises file, showing these two important elements:

```
body common control {
  bundlesequences => {"update", "main", ...};
  inputs          => {"update.cf", "site.cf", ...};
}
```

Executing the promises

The CFEngine3 run consists of executing a given sequence of bundles. And each bundle consists of a collection of promises.

Within a bundle, the promises are executed in a round-robin fashion, according to the normal ordering. If some promises cannot be satisfied on the first try, they are retried again during the next turn of the round-robin, which goes up to the maximum of three iterations. This way the system can hopefully converge to it's final state. If it manages to achieve that in at most three iterations.

Analysis of this process reminds us about something important relating to the nature of promises: CFEngine3's promises do not denote system resources, they denote *actions on* resources. We should rather think of them in categories of calls to high-level functions, than in terms of instantiation of objects.

In other words, declarativity of CFEngine3's promise syntax does not mean, that we declare *what* resources should there be and that we do not care *how* will they be brought into the desired state. It rather means, that we declare *what* actions should be executed and we do not care *how* exactly to execute them on a given host (nor in which order). For example, we order CFEngine3 to "fill this local file X with that remote content Y" instead of giving a sequence of operations expliciting how to connect to the remote server, download data and write it to the given file.

Therefore CFEngine3's real line of work is more or less equivalent to executing all the promises like a sequence of sophisticated high-level scripts.

Default behavior
The default configuration of CFEngine3 implements in fact the "one server – many hosts" scaffolding.

The very first action performed by this basic scaffolding is downloading current versions of all the promise files from the main server. This operation is based completely on the standard CFEngine3 mechanisms. When the CFEngine3 agent is activated, he reads the main promise file and all the logic is inside it: the first promise file included there is `upload.cf` and the first bundle executed is `upload`. So the first thing done is executing the promises contained in the `upload` bundle, which simply copy whole remote `inputs/` directory (containing all the promise files) and overwrite the local one.

This way the standard scaffolding of CFEngine3 emulates the typical behavior of a distributed configuration management tool (i.e. every host *pulls* the configuration updates from the server and applies them to itself in constant intervals of time), using solely promises.

Actors involved
The CFEngine3 installation on every host takes form of several semi-independent modules working together. These are the main components:

`cf-agent` : The part of CFEngine3 which executes the promises and manipulates the system resources.

`cf-serverd` : The server module. Its main role is to share files.

`cf-promises` : The promise verifier and compiler. It is a key module, used by other components to handle the promise syntax.

`cf-execd` : A scheduling daemon. Basically it is responsible for running the agent in constant intervals of time.

`cf-runagent` : A helper module. It can talk to `cf-serverd` on another host and request it to execute his `cf-agent` out of schedule. It can thus be used to simulate a *push* of changes to CFEngine3 hosts (of course only if their set of promises already includes checking for updates).

As all these modules are present on every host, therefore there is no inherent distinction between the CFEngine3 *clients* and *servers* in our environment. Each host can fulfill any role (or both), it is only a question of his promises and the general configuration policy.

A.2 Puppet

Puppet is a very popular and widely used distributed configuration management tool, written in the *Ruby* programming language. It was inspired heavily by CFEngine3, but it features many fresh and important ideas.

Puppet models the configuration of managed hosts introducing the concept of abstract resource layer. One of its particularities is a high focus on declarativity.

A.2.1 Idea

A node's (single machine) configuration state is modelled by a graph of resources (basic units of configuration) with relationships between them. This graph can be represented in form of a manifest (program written in the Puppet's declarative programming language) or in its compiled version: a catalog.

Each node is managed by a puppet agent (a Puppet's client). Configuration for all nodes is stored and managed centrally by the puppet master (the Puppet's server). Puppet agents periodically check in with the puppet master to update their configuration.

In every run, puppet agent from a certain node sends his facts (in formations about his system) to the puppet master; puppet master responds with a catalog describing current desired configuration of this particular node. The puppet agent is then responsible for making their node match the received catalog. He achieves that by syncing (bringing to the desired state) all of the concerned resources in an appropriate order.

A.2.2 Resources

Resource is the fundamental unit of modelling in Puppet. It describes a single aspect of the system. Lists of resource descriptions, written in the Puppet's declarative language, are called manifests. Compiled versions of manifests are called catalogs or (more explicitly) compiled catalogs.

Resource Abstraction Layer

Thanks to the Resource Abstraction Layer (RAL), the high-level models of resources are abstracted from their implementations. The low-level functionality is provided by platform-specific providers. This way resources can be modelled uniformly across different platforms and operating systems.

This homogeneous resource modelization is designed to work only up to a certain level. Even between different *Linux* distributions there are enough idiosyncrasies to surpass the capacity of Puppet's abstraction layer. However, the aim of RAL is not to conceal all differences between platforms, but rather to provide a common interface for all the resources, no matter the platform.

A simple example: *Apache2* HTTP server in *Debian* or *Ubuntu* is known as "apache2" whereas in *Redhat* or *CentOS* as "httpd" (this is both package's and service's name). This difference is not contained by RAL and has to be explicitly declared in the manifest. Nevertheless, both resources are used exactly in the same way and their name is almost the only difference we perceive.

Structure

Each resource has:

- a type. There are quite a few core types, but we can also create our own defined types.
- a title : an unique identifier (needs to be unique only in the given resource type) used to refer to this resource instance.

- a list of ATTRIBUTES with values. The set of possible attributes depends on the resource type. There are also some universal attributes, called METAPARAMETERS, which work with any resource type.

Resources in a manifest

We describe resources in the manifests using PUPPET's declarative language.

Syntax

```
resource =
  type , '{' , title , ':' ,
  { attribute , '=>' , value , ',' } ,
  '}' ;

type      = ? available resource types ? ;
title     = string ;
attribute = ? attributes available for this resource type ? ;
```

Example

```
package { 'openssh-server':
   ensure => installed,
}

file { '/etc/ssh/sshd_config':
   source  => 'puppet:///modules/sshd/sshd_config',
   owner   => 'root',
   group   => 'root',
   mode    => '640',
   notify  => Service['sshd'],
   require => Package['openssh-server'],
}

service { 'sshd':
   ensure => running,
   enable => true,
   hasstatus => true,
   hasrestart => true,
}
```

In this example we describe three resources:

- The software package `openssh-server`, which is `installed`.
- The file `/etc/ssh/sshd_config`: its content is taken from a particular remote file (`source`), it has certain ownership (`owner`, `group`) and permissions (`mode`) settings and it is in relationship with two other resources: the service `sshd` and the package `openssh-server`.
- The service `sshd`. It is currently `running`, it is launched by default on the system start-up (`enable`) and it knows how to handle `status` and `restart` requests.

Core Types

In PUPPET there are currently around 50 native resource types. Some of them are very general and widely used (e.g. file, package, service, user, group, exec), while others are much more specific (e.g. ssh_authorized_key, cron, vlan, mailalias, yumrepo). The three most basic and most important core types are:

- File : manages local files and directories
  (attributes: present or absent, content, ownership, access rights, etc.)
- Package : manages software packages
  (attributes: installed or absent, source, etc.)
- Service : manages running services
  (attributes: running or stopped, enabled or disabled on system start-up, etc.)

Exec

EXEC is one of the PUPPET's native resource types. In fact we should consider it rather as a pseudo-type, as it does not denote any real system resource. Exec has a different purpose: it provides a possibility of executing arbitrary external commands, integrating them smoothly into the PUPPET's resource model. There is one important constraint: if we want exec to work well with PUPPET's resource synchronization mechanisms, commands executed with its help should always be idempotent.

Using the exec resources is supposed to give us a way to work around the PUPPET's native types' shortcomings. In theory this type should not be used unless necessary. However, it is worth noticing, that (at least currently) using the exec resource is often unavoidable even in some quite simple examples.

Relationships

Relationships between resources are declared using four special metaparameters: `require`, `before`, `subscribe` and `notify`:

- Metaparameters `require` and `before` declare a DEPENDENCY RELATIONSHIP:

  $$\{X : \text{require} => Y\} \text{ equals } \{Y : \text{before} => X\}$$

  Both are completely equivalent and mean exactly:

  "Y must be synced before X"

- Metaparameters `subscribe` and `notify` declare a REFRESH RELATIONSHIP:

  $$\{X : \text{subscribe} => Y\} \text{ equals } \{Y : \text{notify} => X\}$$

  Both are also completely equivalent and mean exactly:

  "Y must be synced before X
  and
  X refreshes when Y is changed"

Signification of "refreshes" depends here on the type of the refreshed resource. Currently three native PUPPET's types support refreshing: service, exec and mount. In case of a service resource, refreshing simply means restarting the service. In case of an exec resource, the behavior can be defined; by default it means rerunning the exec's command.

As we can see, the refresh relationship always incorporates the dependency relationship: if X subscribes Y, then X also requires Y.

Autorequires

Certain pairs of PUPPET resources are implicitly put in dependency relationships by a feature called AUTOREQUIRE. This mechanism works only if both concerned resources are explicitly managed by PUPPET (i.e. both have to be declared in the manifest). Therefore, no resources get added to the manifest without our knowledge.

Autorequire is really a quite practical and sensible mechanism: it is just adding some obvious dependencies, which we would have to do add by hand otherwise. For example:

- If PUPPET is managing a file and the user (or the group) that owns this file, the file resource will automatically require his owner.
- If PUPPET is managing a file and any parent directories of this file, the file resource will automatically require his parent directories.

Flexibility, modularity, reusability

In order to facilitate the system administrator's job, PUPPET provides several mechanisms augmenting flexibility of the manifests. Thanks to that, we can write shorter, more readable manifests, which can adapt themselves to different machine types and different conditions. The PUPPET's declarative language lets us:

- Use VARIABLES, CONDITIONALS and FUNCTIONS to make parts of manifest vary with respect to system's facts (facts in the manifests are simply implemented as top-scope variables).

  This lets us, for example, explicit in the manifest, how certain resources should differ with respect to the operating system installed on a particular machine. Moreover, thanks to this mechanism we can avoid rewriting manifests for each single node when we need to include some low-level machine-specific details in the description of certain resources (e.g. IP address, MAC address, hostname).
- Group together some resources in (parameterizable and reutilizable) CLASSES and DEFINED TYPES, which then can be included in manifests.
- Make use of simple yet convenient *Ruby's* template mechanism (*erb*) to create TEMPLATES of configuration files.
- Bundle together our classes, defined types and files (especially configuration file templates) in bigger units known as MODULES. (Using modules is strongly encouraged in PUPPET. They also provide some minor additional benefits, like auto-loading of classes and defined types.)

Defined types

In PUPPET, it is possible to assemble parts of the resource model into more complex parameterizable and reusable structures, creating new resource types. We can do it using the mechanism called DEFINED TYPES.

As the new types created this way must consist of resources of already existing types, it would seem, that this does not give us any opportunity to introduce anything new to the PUPPET's resource model. However, we also have access to the resources of the pseudo-type exec, which offers the possibility of executing arbitrary external commands. Thanks to that, we can de facto step out of the model and design truly new resource types.

Good and simple examples of defined types of this kind are the two types designed to manage *Apache* sites and modules directly as PUPPET resources (through calling the external commands `a2ensite` and `a2dissite` for sites and `a2enmod` and `a2dismod` for modules). They are available as one of the examples of PUPPET's already prepared manifest patterns available on PUPPET's website: http://projects.puppetlabs.com/projects/puppet/wiki/Debian_Apache2_Recipe_Patterns.

Plugins

Defined types are not our only option for creating new resource types. The Puppet's resource model is fully extendable by custom plugins. We can write:

- custom facts
- custom functions
- custom resource types and their providers

All of these need to be written in *Ruby* and integrated into Puppet as modules or parts of modules.

A.2.3 Configuration update run

Actors involved

- Puppet agent is the Puppet's client. An instance of puppet agent runs on each managed node.
- Puppet master is the Puppet's server. It has access to information about the desired configuration of each node.

Data exchanged

Facts Facts are all the important information about a system (which are available normally to the machine's user), put in a convenient and easily consumable form of a list of variables with values. Puppet uses a tool called *Facter*, which extracts system information from various sources (i.e. shell commands and system files) and converts them to the desired form.

Manifests Manifests are lists of descriptions of resources, written in the Puppet's declarative language. They can contain facts, variables, conditionals and references to modules and external files. Hence, certain parts of manifests may evaluate to different values depending on particular external conditions (e.g. the same manifest may evaluate to a different set of resources on Debian and Redhat machines).

Catalogs Catalog (aka compiled catalog) is a DAG of resources, obtained by compiling a manifest in a given environment (which basically means: with a given set of facts). It does not contain any variable parts any more, all of them have been evaluated.

During the process of compilation, Puppet also verifies if the dependencies between the resources does not form cycles (i.e. is the graph of resources a valid DAG).

Configuration run

Every 30 minutes (the default timeout) or on demand, puppet agent checks in with the puppet master, in order to update the configuration of his node:

1. Puppet agent sends the facts about his node to the puppet master.
   (Note: He sends facts and only facts! He does not send any information about the current state of resources on his node.)
2. Puppet master performs the node classification and the right manifest for the given node is retrieved or generated.
3. Puppet master compiles the manifest to the form of a catalog, using the facts provided by the puppet agent.
4. Puppet master sends the compiled catalog back to the puppet agent.

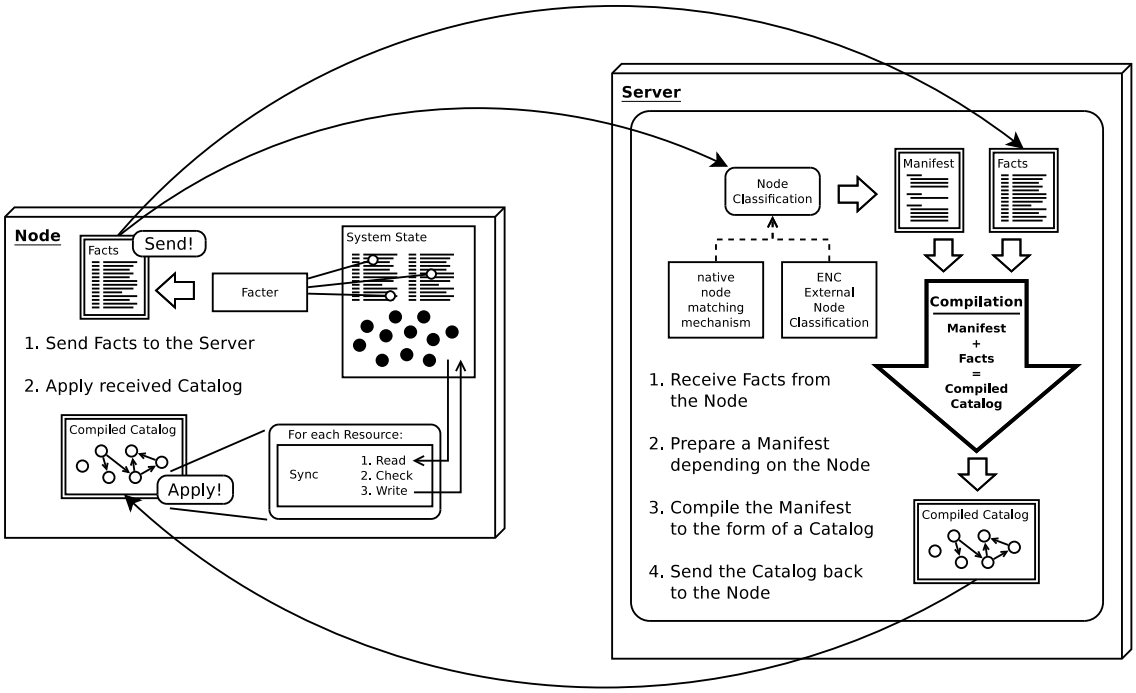


Figure A.1: Puppet configuration run diagram

5. Puppet agent applies the catalog on his node in order to bring it to the desired state.
6. Optional: puppet agent can be configured to send reports to puppet master at the end of every configuration run.

Figure A.2.3 illustrates this procedure, showing how different configuration objects are exchanged and processed.

Node classification mechanisms

Puppet offers two methods of determining, which resources should be included in the manifest for a given node:

1. The native Puppet's node's classification mechanism

   Mapping resources to machines is based on simply matching the machine's hostname with a given string or regular expression. More precisely: there is one main manifest where we explicitly group resource descriptions in node blocks, and the right block is chosen using this matching method. A built-in authentication system based on certificates takes care of the security issues.

2. External Nodes Classifier

   We can designate an arbitrary program, which Puppet will run in order to acquire the description of resources for the given node. This description has to be provided (by the program's output) in a specific YAML format and the corresponding manifest itself is generated using it.

Applying the catalog on the node

A puppet agent applies the catalog is to his node by syncing all the concerned resources. Order of syncing is induced by the relationships between resources.

A single resource sync operation consists of three steps:

1. Read : read the current state of the resource in the system.
2. Check : compare the current state with the desired state.
3. Write : make necessary changes (if any).

It is important to notice, that the application of a catalog is in fact a quite straightforward operation. There is not much verification performed here (instead we have tools, which permit us to simulate the whole process without of making real changes to the system). We can easily imagine catalogs, whose application results in errors or never ends, because of some details happening below the Puppet model's level of abstraction. For example, the real software packages dependencies, on the package manager level, may mess up the Puppet package resources dependencies, that we have declared on the manifest level.

Simplifications

This is only a simplified view of the Puppet's configuration run. In reality the compilation process consists of several phases and a few semi-compiled forms of the manifest are created in the meantime. Moreover, some part of the compilation (a phase called instantiation) has to be performed already on the client, as the server does not have enough information to handle it.

It is also not exactly clear, when the final form of the compiled catalog is really obtained and when its validity (the correctness of the resource DAG's structure) is verified. But as Puppet is an open source project, all these details can be without doubt clarified by reading the code.

A.3 MCollective

MCollective is a framework to build server orchestration or parallel job execution systems, written in the *Ruby* programming language. Basically it is a kind of middleware. MCollective was bought by *Puppet Labs* and integrated into the Enterprise edition of Puppet in order to enhance it with some multi-server management and orchestration capabilities.

MCollective does not really change anything in Puppet itself. It creates an additional layer above the standard Puppet, which lets us control resources and machines on a slightly higher level.

A.3.1 Idea

On every node (a machine) a MCollective server is running. Server is managing the connection to the middleware (a publish-subscribe communication system) and hosts agents (blocks of code which perform specific roles).

A MCollective client can produce messages containing commands for remote agents. These messages are distributed via the middleware to all the nodes. Messages can contain filters. Nodes, which do not satisfy the filter, simply ignore the message. These, which satisfy the filter, act upon receiving the message and dispatch the message to the right agent. The agent then processes the message and possibly prepares a reply. All the replies from the nodes are returned to the client. This message flow system is a version of the broadcast paradigm.

A combination of all the servers, nodes and middleware operating in the same namespace (a publish-subscribe middleware's "topic") is called a collective.

A.3.2 Concepts

Node A node is a machine on which a MCollective server is running. A node is *never* identified by his address or hostname, but only by his metadata. This approach comes from one of the main MCollective's principles: real-time discovery of the network resources.

Various sources of metadata can be used. Currently Puppet and *Chef* are supported natively. It is possible to add the support for other sources through plugins (*Facter* and *Ohai* are already supported this way).

Server A server is a MCollective daemon who is responsible for:

- Hosting the agents.
- Managing the connection with the middleware through a component called connector.

Standard connector works well with *Apache*'s *ActiveMQ* middleware, but we can replace the middleware (and the connector) if we want.

Client A client is a program who sends the messages and handles the replies from agents.

Agent An agent is a piece of code – a *Ruby* class. It has two basic characteristics:

- It is capable of handling an incoming message (it has a `handlemsg` method).
- It has a name (his unique id on a node) and some other metadata.

Basically a server who receives a message is responsible of passing the body part of the message to the right agent (which means calling his `handlemsg` method with the body as a parameter). This mechanism is depicted on the figure A.3.2 on the next page.

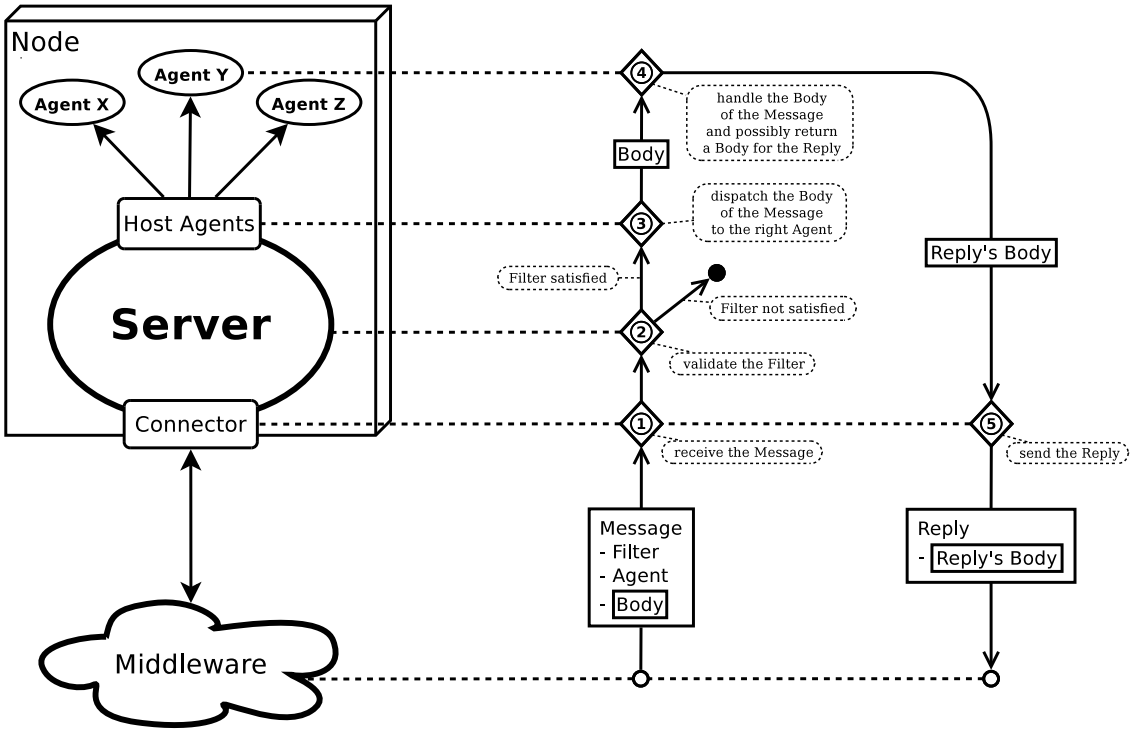


Figure A.2: MCollective server diagram

Broadcast Paradigm

The message flow in MCollective implements a variant of broadcast paradigm. It works in the following way (illustrated on the Figure A.3.2 on the following page):

1. The client sends a message (only a *single instance* of a message!) with a filter attached.
2. The middleware broadcasts the message to all nodes in the collective.
3. Every node gets the message and validates the filter.
4. Only nodes which satisfy the filter act on the message and (optionally) send replies.
5. The middleware passes all the replies back to the client.

This solution has important advantages, but also some drawbacks. As we can see, the work of filtering the messages is not centralized on the server, but distributed between all the nodes, which permits MCollective to scale up very well. On the other hand, this results in a big number of messages which need to be sent and received (in comparison to a solution with central filtering of messages), which may in certain situations greatly increase the load of the network.

Message format

The main parts of a message are:

- the name of the collective to which the message should be broadcasted,
- a filter,
- the name of the agent who should handle the message,
- the BODY – the data which should be passed directly to that agent.

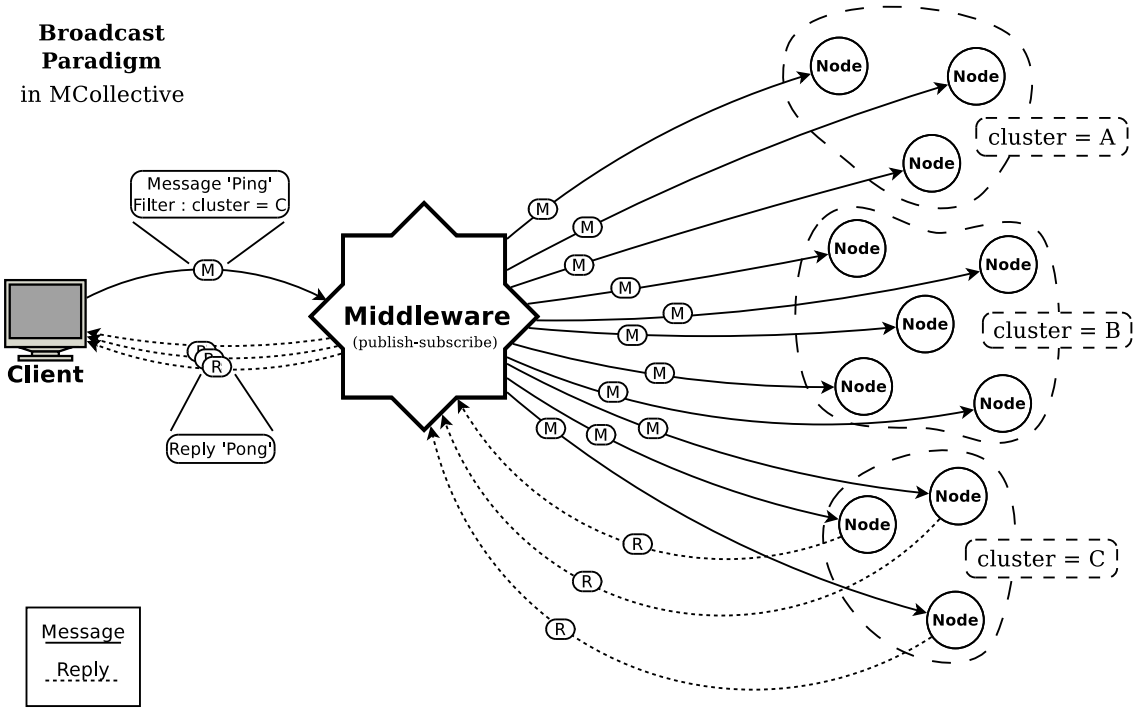


Figure A.3: MCollective broadcast paradigm diagram

A.3.3 SimpleRPC

SimpleRPC is a *remote procedure call* framework built on top of MCollective's native clients, agents and messages. It facilitates the task of creating new clients and agents by abstracting a big part of complexity and providing us with some simple conventions, a data definition language and a fine grain authorization system.

Conventions

Every agent exposes actions: tasks, which he can execute. Each action is a remote method, it has a name, it can require inputs and it can return outputs. Actions can have quite a lot of meta data attached to them.

When writing an agent, we use the data definition language to define what actions he offers, what inputs these actions take and what outputs they generate. For each input and output we can precise some additional parameters: description, type, input validation, hints about how they should be displayed in a terminal, etc.

SimpleRPC Messages

In order to support SimpleRPC, the body part of a MCollective message is given an additional internal structure. This structure is different for SimpleRPC Requests (messages sent to agents) and SimpleRPC Replies (replies from agents):

- the body of a SimpleRPC Request contains:
  - name of the action which should be performed by the agent,
  - a hash of variables with values, which should be passed to the agent as the inputs for the action.
- the body of a SimpleRPC Reply contains:

– *statuscode* and *statusmsg* which are used to propagate and handle errors,
– a hash of variables with values, containing the outputs of the action, which should be passed to the client as the reply.

A.3.4 Integration with Puppet

MCollective is a component of *Puppet Enterprise*. It is not used in Puppet directly, its purpose is rather to orchestrate big systems managed by Puppet. According to *Puppet Labs*, MCollective:

- Brings multi-server and multi-datacenter orchestration to Puppet.
- Improves how Puppet can work with managing applications on existing systems.
- Simplifies how users schedule complex sequences of activities using data available from the Puppet platform.

Even though this description is a pure marketing text containing some unclear statements, it seems to be quite accurate in characterizing MCollective's contribution to Puppet. Especially if we focus not only on features enumerated here, but also on what is *not* mentioned.

Mainly, MCollective augments our capabilities of exercising central control over hosts governed by Puppet. It allows us to monitor and manage Puppet resources in a multi-server environment in a simple way and it makes possible carrying out sequences of orchestrated configuration modifications. In other words, with MCollective we perceive our distributed configuration as a whole, all the machines and all their resources are on one big picture. Moreover, we can act on it on a global scale, performing coordinated operations on multiple machines at a time.

What MCollective does not do is introducing any kind of direct host-to-host interactions or dependencies. The only possible direction of the flow of control is outward from the decision center (i.e. our workstation) to all the hosts. Therefore, there can be no direct orchestration of two services (living on two separate hosts), negotiating configuration parameters directly between themselves.

Furthermore, any sequence of coordinated actions performed on our environment has to be explicitly scheduled by us. There is no automatic planning or high level request mechanism involved. We simply design a kind of imperative step by step program of desired changes in the distributed configuration, detailing modifications of every resource. However, thanks to the MCollective capabilities, we can easily use the distributed information and metadata to write actions and express conditions concerning whole groups of machines.

Scenarios of use

These are four examples of scenarios showing how MCollective can be used to add some service orchestration to Puppet. We introduce here three types of machines:

- web servers are responsible for handling requests coming from the internet,
- the database server (here we will always have at most one database server) contains a database from which web servers fetch information required to respond to the requests (e.g. to generate web pages),
- load balancers redistribute the incoming traffic between all the web servers (or just a certain group) in order to equilibrate their load.

All the presented scenarios feature a sequence of reconfiguration steps, which have to be executed in a certain order. Most of them also use the MCollective capabilities (i.e. broadcasting messages with filters) to execute certain steps in parallel on multiple hosts.

Scenario 1: a simple coordinated update (or maintenance).

Situation We have just two machines: a web server and a database server. We need to perform an update on the database server and during the time of that update the web server should not be using the database.

Solution

- First we shut down the web server, simply by changing the state of his `apache2` service resource from `running` to `stopped`.
- Then we proceed with upgrading the database server configuration, for example changing the required version of his database software package resource. This results in an automatic restart of the database service.
  (For the purpose of making this scenario more serious, we can imagine, that we need to perform a more complicated maintenance operation, which makes the database state inconsistent for certain time. Thus shutting down the web server is necessary to avoid it serving information from an inconsistent database.)
- After that, we bring the web server back up, putting his `apache2` service resource again in the `running` state.

Although this scenario would be also possible to implement on standard PUPPET, in that case we would need to wait between executing each step in order to give the PUPPET clients enough time to perform a full run and synchronize all the resources. Alternatively, we would have to force by hand a run on each machine in the right moment, simulating a push of changes.

Scenario 2: a multi-server coordinated update.

Situation We have many web servers and a single database server. We still need to perform an update on the database server in the same conditions, but this time we need to take care not of one, but of multiple web servers.

Solution This scenario works almost exactly as the previous one, but with one important exception. We want the actions performed previously on a single web server to be carried out across all our web servers. In order to achieve this effect we simply set a MCOLLECTIVE filter on all the involved messages, making them valid only on machines which contain a `apache2` service resource. And that is it: thanks to the MCOLLECTIVE layer, our algorithm from the first scenario (more exactly: its parts concerning stopping and bringing back up the `apache2` service) would be now executed in parallel on all the web servers, no matter how many of them we have.

Scenario 3: update in two batches in order to not break the service.

Situation This scenario is a little different. We have some load balancers and many web servers (we do not care about the database server in this example). Initially the load balancers are configured to distribute the incoming traffic equally among all the web servers. We need to update each of our web servers. There is also an additional condition: at any given time the service has to remain available, so at least one server has to be running. And we assume, that during the time of being updated, a web server is stopped at least for a short moment needed to restart the `apache2` service (but we can easily imagine a more complicated maintenance operation with the service stopping for a longer period of time).

Solution In order to fulfill these conditions, we divide all our web servers into two (separate) groups. We execute following steps for first group and then for the second group:

- We remove all the web servers of the current group from the load balancers configuration. Thanks to that, no more connections from the internet will be redirected to our current group, all the traffic will go to the other machines.
- Then we shut down these web servers, update them and bring them up again.
- After that we add all the servers from the current group back to the load balancers configuration. And we proceed to perform the same steps for the second group.

Just like in the previous scenario, everything is executed in a parallel fashion throughout our whole infrastructure. Each step happens on all the concerned machines at once.

Scenario 4: update by small groups in order to not break the service and do not increase the web servers load significantly.

Situation Again we have some load balancers and many web servers. But now we do not only need to keep the service available, we also want to not overload it, which may happen if at any given time there are too few web servers running. For example switching half of them off is definitively not an option.

Solution This scenario does not differ too much from the last one. In fact we can execute exactly the same steps, just for different groups of machines. Instead of dividing all machines in two, we perform the operation in groups of 10 servers (imagining our pool of machines contains several tens or hundreds of them), scheduling update on one group every 15 minutes. This way the whole process will be distributed in time and should progress smoothly, without increasing the load of our web servers too much.

Conclusion

As we can see, even the basic possibilities offered by MCollective to a system administrator look very interesting and useful in practice. Scenarios like these would be very difficult to execute on a plain Puppet (which is simply not designed to handle multi-step actions) and yet they seem easy and natural when we have MCollective.

We just need to remember, that everything described above would need to be explicitly written down as a kind of distributed reconfiguration scripts. There is not much declarativity involved here and we would have to describe precisely and in the right order all the changes, which have to be performed on all the involved resources. For example, as it is not possible to declare a dependency between a web server service working on one host and a database service working on another host, we need to keep that relationship in mind ourselves and order the actions properly (always stopping the web server service before stopping the database server service, etc).

In fact MCollective is just what it is: a framework for parallel execution of tasks. This framework, applied to Puppet, is doing its job very well, but obviously it cannot be used to surmount all the restrictions of Puppet's resource model.

A.4 Chef

Chef is a distributed configuration management tool created by *Opscode* and written in *Ruby*. It profits from Puppet's experience, but tackles the configuration management problem from a different angle, replacing the pursue of declarativity with a completely imperative approach.

Some people say, that Chef is supposed to be "Puppet done right". In reality it is rather "Puppet made imperative". From the very practical point of view (i.e. according to the community of people using distributed configuration management tools), the difference between these two seems to be mostly a matter of taste.

Another important feature of Chef is introduction of some service orchestration mechanisms, which give it a similar level of capabilities to Puppet expanded with MCollective.

A.4.1 Idea

Our whole infrastructure consists of multiple nodes and one server. The server is the central store of the infrastructure's configuration data. Nodes are the hosts whose configuration is managed.

A node's system state is modelled as a collection of resources, each representing a single aspect of the configuration. In order to operate on the resources, we write recipes: sequences of resource descriptions. Recipes are parameterized using attributes (highly structured configuration variables) in order to make them versatile and suitable for different nodes.

In Chef there is a whole hierarchy of levels of configuration description. Configuration data and configuration artifacts (like file templates) are distributed between objects such as cookbooks, data bags, roles, environments and nodes themselves. This rich collection of configuration objects gives us a possibility of defining several layers of configuration patterns: from the default general configuration which should apply to every node, through role-specific configuration which should apply only to certain groups of nodes (like database servers or load balancers), down to the node-specific configuration which should apply only to a given host.

Chef is designed with the "thick client – thin server" approach. This means that nearly all the work is done on the node's side and the server is used almost exclusively as the configuration data repository.

Another crucial Chef's principle is sticking strictly to the imperative paradigm, in place of joining the pursuit for declarativity, common for other configuration management tools. This results in adopting the order matters philosophy: a decision to always make the ordering of performed actions completely explicit.

Note: Chef is quite difficult to understand at first, mostly because of the rich repertoire of various forms of configuration data (and the complex ways how they work together) and as a result of the particular syntax of its resource description language, which is completely imperative, but mimics some features of declarativity.

A.4.2 Resources and Providers

Resources

Resource is a fundamental unit of work in Chef. It represents a cross-platform abstraction of a single aspect of the host's configuration (e.g. file, package, service).

Each resource is a proxy for a piece of the system state. We can affect that piece of system state indirectly, by performing actions on the resource (e.g. create, install, upgrade, start, stop). Thanks to the mechanism of providers, these resource-level actions are then translated to real system-level operations, which are carried out on the host.

Resources have attributes (e.g. name, owner, path). The attributes are the pieces of data, that a resource contains, structured as a list of nested key-value pairs.

Depending on a resource, there are different actions and attributes available. There is a group of common actions and attributes available for all resources (called, unsurprisingly, common

ACTIONS and COMMON ATTRIBUTES).

Note: In CHEF, the concept of "resource attributes" (described above) is something completely different than the concept of "attributes" in general (they have a similar structure, but are used in a completely different way). At some point in CHEF's development there was an initiative to rename the "resource attributes" to "parameters" in order to avoid confusion, but finally this change did not happen. Therefore, from this point on we will always clearly distinguish if we speak about resource attributes (in contrast to simply: attributes).

Providers

PROVIDER is a platform-specific implementation of the aspect of configuration, which a resource abstracts over. Each provider is responsible for bringing the system part corresponding to a resource into the state described by this resource. In order to achieve it, a following sequence of events happens:

1. The provider is given a resource of a right type: this resource represents the desired state of a specific system part and our provider knows how to perform operations on this system part.
2. The provider looks up the current state of this system part on the host.
3. The provider performs the action specified in the resource's description and in this manner brings the system part into the desired state.

Every type of resource can potentially have many different providers available: each one designed to work on a different platform. It is similar to a typical abstraction-implementation pattern: an abstraction (resource) can have many different implementations (providers).

A.4.3 Configuration description

Recipes

RECIPES are CHEF's fundamental configuration description units. A recipe encapsulates a sequence of resource descriptions. All the resources in a recipe are described using the CHEF's Domain Specific Language (DSL), which is an extension of *Ruby* (in fact recipes are executed directly as *Ruby* code). The order of resource descriptions in a recipe is important, as they are evaluated exactly in the provided sequence; this embodies the order matters philosophy.

It is important to notice, that in spite of its particular meaning, the action, which has to be performed on a resource, is expressed in the CHEF's syntax exactly in the same way as if it was one of the standard resource attributes.

All the resource types have a default action, which is performed if no action is declared explicitly. Often the default action is simply doing nothing.

Single resource in DSL

Syntax :

```
resource =
   type , name , "do" ,
   { attribute , value } ,
   "end" ;

type      = ? available resource types ? ;
name      = string ;
attribute = ? attributes available for this resource type ? ;
```

Example :

```
user "sam" do
     action :create
     home "/home/sam"
     shell "/bin/zsh"
     comment "Sam is happy."
end
```

Recipe example

Recipe `ntp.rb` :

```
package "ntp" do
    action [:install]
end

template "/etc/ntp.conf" do
    source "ntp.conf.erb"
    variables( :ntp_server => "time.nist.gov" )
end

service "ntpd" do
    action [:enable,:start]
end
```

In this recipe example we introduce three resources and we define a sequence of actions, which should be performed on them exactly in the given order:

1. First the package `ntp` should be installed.
2. Then a file `/etc/ntp.conf` should be created using a *Ruby* parameterizable template `ntp.conf.erb`. The template code should be executed in an environment with a variable `ntp_server` set to a string `"time.nist.gov"`. As `create` is the default action for the template resource, it does not have to be indicated explicitly.
3. Finally, two actions should be executed for the service `ntpd`: it should be enabled (set to be started automatically on boot) and then it should be started (right now).

On the declarativity of Chef

The mechanism of defining resources in Chef's DSL seems declarative, but we should keep in mind, that in reality we are dealing with a completely imperative language. The syntax is mimicking declarativity very well, so we have an impression of writing a catalog of abstract resources. But in practice our declarations are executed one by one, similarly to imperative programming instructions for instantiation of objects.

However, some degree of declarativity is genuinely present in Chef, as the actions on the resources described in the recipe are not simply carried out right away when the recipe is loaded. Chef has distinct compilation and execution phases. When a resource in a recipe is evaluated, it is put in a special array called RESOURCE COLLECTION. And only after all recipes are loaded and everything is prepared (the compilation phase is finished), the whole sequence of actions on resources from the resource collection gets executed one by one (exactly in the order of putting them inside).

As we can see, this whole mechanism is focused rather on the actions performed on resources, not on the abstract resources themselves (i.e. their state). That is why it is possible, for example, to assign more than one action to a single resource.

Attributes

Attributes are a highly structured form of configuration data used in many of Chef configuration objects. Their purpose is to parameterize recipes.

Structure Attributes take form of nested key-value pairs, which is equivalent to a tree with labeled nodes and values on the leafs (lists of values are also allowed). This is an example of how attributes are structured:

```
attributes
├─ key1 :  value1
├─ key2
│  ├─ key3 :  value3
│  └─ key4 :  value4
├─ key5 :  value5
└─ key6 :  [value6, value7, value8]
```

Merging During a Chef's run, attributes from many sources (cookbooks, environments, roles and nodes) are merged together, through a sequence of operations called *deep merges*. The aim of this process is more or less to maintain several layers of configuration settings (from more general infrastructure-wide defaults down to the node-specific settings), where the lower layer inherits all the attributes from the higher layer and can leave them as they are, override them and/or add new ones.

The operation of merging of attributes always follows a specific order, which depends on the order of the sources of the attributes. A good explanatory example of how this mechanism works can be found in Table A.1 on the next page.

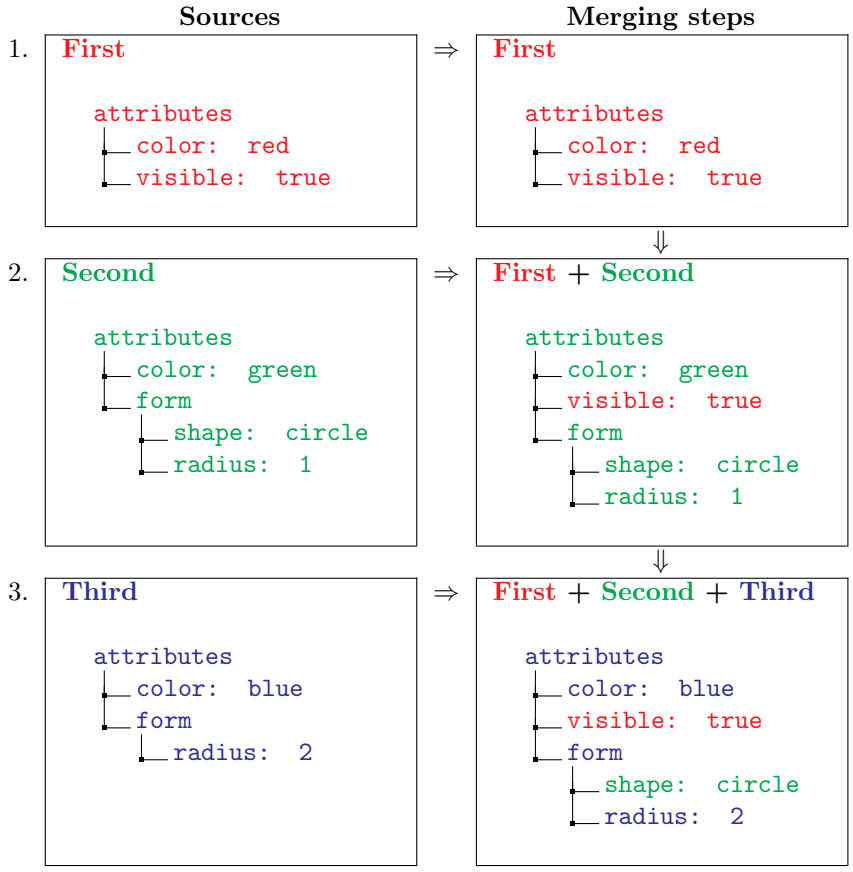
Types In order to control and customize the attributes' merging process, we can declare an attribute as one of several types, which have different precedence level and behave in a different way during Chef runs:

- Default attributes behave exactly as we expect: if a default attribute is declared on a higher level and then redeclared on a lower level, the second declaration overrides the first one.
- Override attributes from higher level do not get overridden on the lower levels, unless the same attribute is redeclared on the lower level also as an override attribute.
- Normal attributes behave just like default attributes, but are saved between the Chef runs. They also have higher precedence than default attributes, so they are available in the next run if not overridden by override attributes.
- Automatic attributes are a special type of attributes, they cannot be declared manually and they always override all the other types. As they represent facts about the system (usually discovered by some tool able to retrieve this kind of information on various platforms and return it in a consistent form, like *Ohai* or *Facter*), this behavior is completely understandable, because trying to redeclare them would be as pointless as trying to change the truth about the given host (e.g. pretend that a host is running *Windows*, not *Linux*).

Notation Attributes manifest themselves in three forms:

- they can be defined (in attribute files) and accessed (in the recipes) directly as *Ruby* data structures,
- they can be stored and transferred in a form of *JSON* code,
- they can be defined in the meta-data of various objects, like roles and environments.

Table A.1: Example of merging attributes from three sources.



Examples

- Declaration in an attribute file.
  (In the attribute declaration we need to specify its type, because the declaration "happens" before the merging process. Here, for example, we declare a default attribute.)

```
default["apache"]["listen_ports"] = [ "80","8080" ];
```

- Accessing in a recipe.
  (If we access the attribute like that, all the layers of attributes have been already merged at this point, so we obviously do not distinguish between different types.)

```
x = node["apache"]["listen_ports"]
```

- *JSON* form:

```
{ "apache": { "listen_ports": ["80", "8080"] } }
```

Cookbooks

Cookbooks are the fundamental units of configuration distribution (and sharing) in Chef. Their purpose is to encapsulate configuration data, configuration artifacts and generally all the resources needed for one particular fragment of system configuration, like a single package or service (e.g. ssh daemon, database, web application). Besides, they contain some meta-data, which allows us to manage the cookbooks themselves and define their interdependencies.

Contents The cookbooks can contain a whole spectrum of Chef's configuration data objects and configuration artifacts:

- recipes (there must be at least one recipe: `default.rb`)
- attribute files
- template files
- cookbook files (i.e. any standard configuration files, which are not templates)
- resource definitions (simple reusable definitions of new resource types)
- custom resources and providers (created using standard plugins mechanism or the Lightweight Resources and Providers)
- libraries (containing arbitrary *Ruby* code, they allow us to extend the DSL, plug in to existing infrastructure etc.)

Structure Cookbooks are usually stored on disk in a directory structure which reflects their content's organization:

```
cookbooks
└── [cookbook name]
    ├── attributes/
    ├── definitions/
    ├── files/
    ├── libraries/
    ├── providers/
    ├── recipes/
    ├── resources/
    ├── templates/
    ├── metadata.rb
    └── README.rdoc
```

Meta-data Cookbook's meta-data contains:

- its name (which is also easily inferred from the directory name),
- its version (in Chef, when we refer to cookbook versions, we can always use standard mathematical operators to precise the version number),
- information, how it depends on other cookbooks (this dependency mechanism is quite basic and serves only to make sure, that all the required cookbooks in proper versions are downloaded and loaded together).

Templates

Templates are files written in a markup language, that allows to dynamically generate the file's final content based on some variables. Chef uses the embedded *Ruby* (erb) templates and lets us instantiate them through the normal resources syntax.

There exists also a special Template File matching mechanism. It permits us to create several version of one template file: a default version, versions for different platforms and versions for specific hosts. When the template is used, the version most appropriate for a given host is chosen and applied.

Example

Instantiation of a template file:

```
template "/etc/abc.conf" do
   source "abc.conf.erb"
   variables (
      :x => 1
      :y => 2
   )
end
```

Simple Resource definitions

DEFINITIONS allow us to create new, parameterizable resources. In theory, it lets us customize and encapsulate one or more resources and then instantiate them multiple times, as if they were a single resource. In reality, because of CHEF DSL's imperative nature, this mechanism works more like if we were simply defining and executing procedures with parameters.

Syntax

```
definition =
  "define" , resource_name , { ',' , parameter } , 'do' ,
  definition_of_a_resource ,
  'end' ;

resource_name = ':' , string ;
parameter     = ':' , string ;
```

Example

Definition A sample definition of a new resource:

```
define :happy_user, :shell do
   user params[:name] do
      action :create
      home "/home/happy/" + params[:name]
      shell "/bin/" + params[:shell]
   end
end
```

Execution How we could instantiate the defined resource:

```
happy_user "sam" do
   shell "zsh"
end
```

Result How we can imagine the actual resource, which is created by our code:

```
user "sam" do
   action :create
   home "/home/happy/sam"
   shell "/bin/zsh"
end
```

Lightweight Resources and Providers

Definitions allow us to create new pseudo-resources by bundling together some existing elements. However if we need some truly custom resources and providers, we have to write them as Chef plugins.

Normally it is not an obvious task: we have to not only know *Ruby* quite well, but also understand how to properly write a working Chef plugin (using the correct classes in the right way, etc). But here comes to aid a special mechanism, designed to allow us to create new resources and providers in a simplified fashion: Lightweight Resources and Providers (LWRP). Without entering into details: LWRP is something like an additional Domain Specific Language in Chef, dedicated to writing custom resources and providers. It is an extension of *Ruby*.

Existence of LWRP in Chef is worth noting, because it is quite an uncommon concept for configuration management tools: an interesting method to introduce something between resources native to the tool and typical crude mechanism of plugins.

Nodes, roles and environments

These three types of objects: nodes, roles and environments, together let us define the configuration of all the hosts in our infrastructure in a very flexible and sophisticated way:

- A node is a single host that is managed by Chef.
- Roles provide means of grouping similar features of similar nodes. We use roles to express the parts of the configuration that are shared by a group of nodes. They let us easily compose sets of functionality.
  Examples: application server, web server, database server, etc.
- Environments provide a mechanism for managing different architectural segmented spaces.
  Examples: production, staging, development, testing, etc.

Roles and environments are two concepts which seem to be quite similar, but they have different goals. Roles are meant to group hosts that perform similar functions. Environments are for other types of administrative grouping. Roles can span environments, i.e. when we define a role, we can make its configuration vary with respect to the environment. For example, we can want our application servers ("application server" is obviously a role) to connect to the development database when they are in the "development" environment and to the production database when they are in the "production" environment.

Node's and role's structure The primary features of a node and a role are the same:

- Attributes : which are merged with attributes from all the other sources during Chef's run,
- Run list : list of recipes to run and roles to include.

Run lists A run list in the simplest case is a list of recipes that a node will execute. The order, in which recipes are listed in the run list, is exactly the order, in which Chef will run them.

When we want to put a recipe in a run list, we specify it by giving its cookbook name and optionally the recipe's name. If we just precise the cookbook name, the default recipe (`default.rb`) is used.

Roles also have their run lists. Moreover, we can include roles in run lists. If we put a role in a run list, then during the execution the role simply gets expanded to this role's whole run list.

Additionally, when we include a role in a node's run list, we indicate, that this node belongs to that role. Which means (among other things), that the role will become one of the attribute sources for the recipes executed on this node.

Example

```
{
   recipe["ntp"],                 # runs the default recipe
                                  # from the cookbook "ntp"

   recipe["mysql"::"server"],     # runs the recipe "server"
                                  # from the cookbook "mysql"

   role["application server"]     # expands to the run list
                                  # of the role "application server"
}
```

Environment's structure Environments work in a slightly different fashion than nodes and roles:

- they also contain attributes,
- but they do not contain run lists. Instead, in nodes and roles we can define multiple run lists, a different one for each environment.
- Besides, environments let us specify version constraints on cookbooks: decide which versions of a cookbook should be used in a given environment.

Data Bags

DATA BAGS provide a globally available data store. They serve a similar role to that of attributes, but on the level of the whole infrastructure. Moreover, they can be not only accessed directly and modified, but also searched.

Structure and organization We can have as many data bags as we want. Each data bag can contain any number of data bag items. We can also have some "global" (or "common") items, which do not belong to any bag.

DATA BAG ITEMS have the same structure as attributes: nested key-value pairs. They are stored in the form of *JSON* files on the server, in a directory structure reflecting their organization:

```
data_bags
├── data_bag_a
│   ├── item_a1.json
│   └── item_a2.json
├── data_bag_b
│   ├── item_b1.json
│   ├── item_b2.json
│   └── item_b3.json
├── global_item_1.json
├── global_item_2.json
└── global_item_3.json
```

Usage We can access (and modify) data bags in recipes either directly or through the search feature. The contents of data bag items are imported into the recipe and manipulated in a form of *Ruby* associative arrays.

Data bags can perform two roles:

- They are effectively a kind of shared memory (shared between all the nodes in our infrastructure), with all its advantages and inconveniences. They can be used as a communication mechanism, which can be applied to various tasks, for example to attain some degree of

service orchestration. We could easily imagine a coordination between application servers and database servers, based on exchange of information (like database name and password) through a data bag.

- They can be used for storing configuration decisions in a centralized fashion to permit *data driven* configuration management. For example, we could maintain the list of users in a data bag and make all machines synchronize their users with this list on each Chef's run.

Search

One of Chef's particularities is the presence of a global search mechanism for the whole infrastructure. Complete information about all the nodes and their current configuration data is maintained on the server and kept up to date at all times.

Every host, at the end of each Chef run, saves his state at the server. Thanks to that, we can use the search engine to query information about the infrastructure. And there is quite a lot of searchable data available: node attributes and run lists, roles, environments and data bags. For example we can easily find things like:

- all nodes which have *Debian* installed,
- all nodes which have the role "application server" and are in the environment "production",
- all roles which execute the recipe for *Apache* in their run list.

The Chef's search feature allows us to introduce capabilities of *distributed discovery* to our infrastructure. When executing its recipes, a node can actively query the information about other nodes and hence make decisions based on the current state of the infrastructure and its components.

A.4.4 Chef Run

Chef run is a single execution of the Chef client on a node. The main idea behind this process is *convergence*, defined in Chef as "bringing the system closer to correct with each action you take". Every Chef run should bring the system into the desired state (or at least closer to it).

Phases A Chef run consists of five major phases:

1. Build the node (also register it with the server and authenticate)
2. Synchronize cookbooks
3. Compile the resource collection
4. Configure the node
5. Run notification handlers (not important to us here)

Build the node

At the beginning, the Chef client running on a host constructs the node object, using configuration data from several sources:

1. the previous node data fetched from the server,
2. additional attributes and recipes (which can be introduced for example through the command line),
3. system facts discovered by *Ohai*, converted to the form of automatic attributes.

Synchronize Cookbooks

Then the client downloads from the server the whole collection of cookbooks (with all their content: recipes, template files, libraries, etc.) and stores them in the local cache. Of course Chef is trying not to download everything every time, but only to synchronize the cookbooks it needs.

Compile the resource collection

Now the Chef's client has all the configuration data he needs. He begins to load it all and to build the resource collection: an array of all the resources which should be executed on the node.

This process obeys a specific order, which follows approximately this way:

1. Libraries : First we load all the libraries from all the cookbooks, to make available the language extensions and other *Ruby* objects defined inside.
2. Attributes : Then we load all the attributes form various sources and merge them together in a specific order.
3. Definitions : After that we load all the resource definitions (which create new pseudo-resource types) and all the custom providers and resources (written as standard plugins or defined using the Lightweight Resources and Providers).
4. Recipes : Finally we load all the recipes (one by one, following precisely the order defined in the run list) and we put each evaluated resource in the resource collection. Of course, when we load a recipe and evaluate the resources defined inside, the plain *Ruby* code outside the resources gets evaluated as well.

Note: It is interesting to know, that even though in Chef we have a separate compilation and execution phases for resources, we can circumvent the normal flow. We can force execution of some resources already in the compilation phase. Moreover, we can do the same with the free *Ruby* code (the code outside resource definitions): normally it is executed during compilation, but we can wrap it with special resources called ruby blocks, which will be evaluated in the execution phase along with all the other resources.

This mechanisms manifest very clearly the Chef's biggest advantage and drawback: the high flexibility, which may easily lead to high complexity and chaos.

Configure the node

After completing these preparation steps, the Chef client finally has all what he needs to converge the host's system to the desired state.

Each resource in the resource collection is mapped to its corresponding provider. Then each provider performs the action defined in his corresponding resource, thus bringing a specific system part into the right state. This way all the parts of the system, abstracted by our resources, get configured one by one.

When this process is finished, the client saves the current state of the node on the server. This way he persists the node's data (for the next Chef run) and keeps the search feature up to date.

A.5 juju

JUJU is not a classical distributed software configuration management tool, but rather a framework for deploying and orchestrating services in a cloud. Therefore, it approaches the problem of distributed configuration using a particularly high perspective: it is focused on the level of controlling whole distributed applications. Managing the configuration of underlying software components is treated by JUJU in a very simplified way.

A.5.1 Idea

SERVICES (applications) are deployed in the ENVIRONMENT (a cloud) by using CHARMS (descriptions of a service, containing a recipe how to deploy and manage it). Each service is deployed as one or many homogeneous SERVICE UNITS (which can be easily added or removed in order to scale the application up or down). Services are involved in various RELATIONS, which permit them to work together (technical details of properly configuring and managing a certain relation are handled by charms).

A.5.2 Concepts

Environment

ENVIRONMENT is a configured location where we can deploy services.
JUJU is designed to be able to work with any environment which can provide:

- a new MACHINE with *Ubuntu Cloud Operating System* on demand,
- a permanent storage facility.

However, currently JUJU works only with *Amazon EC3* machines and *Amazon S3* permanent storage. Possibility of using other cloud solutions is a future functionality.

A diagram illustrating an organization of the JUJU environment and interactions between its components is shown on the figure A.5.2 on page 82.

Service

SERVICE is an application (or set of applications) integrated into the framework as an individual component (e.g. mysql, wordpress, mediawiki).

- JUJU service = external application + its charm.
- We can customize service's settings and behavior by changing the SERVICE CONFIGURATION.
- Each deployed service is distinguishable and has a unique name. Inside of one environment we can deploy multiple services which are based on the same charm (e.g. `wordpress-blog-of-Alice` and `wordpress-blog-of-Bob`).

Service Unit

SERVICE UNIT is a single running instance of a service.

- All service units of one service always share the same:
  - charm
  - relations
  - service configuration
- Correctly prepared charm should allow us to scale the service up easily, just by adding new service units to it, and to scale it down, by removing some of the existing service units.

- Currently we can deploy only one service unit per machine. JUJU developers announced that deploying multiple service units per machine should become possible soon. However, there are some issues in JUJU, which make us a little skeptical in regard to this future functionality.
- All service units within a single service share the same charm and configuration. From the point of view of the framework, they are created identical, they all perform the same role and they are completely homogeneous.

  There is a possibility to work around this restriction: we can program the charm in a way that permits service units of one service to have different roles. However, as we always have exactly the same charm and the same relations deployed for all of them, the role of each service unit has to be somehow determined in the run-time by the service unit itself (i.e. they have to self-organize themselves, because from the point of view of the framework they always stay equal). So doing that means working around the JUJU's model.

  It is worth noticing however, that there exist already some well established examples of charms, which use this technique. For instance, the `mysql` charm, found in the JUJU charm repository, uses kind of a dirty trick to distinguish between the first launched service unit (which becomes the database master) and all the subsequent ones (which become database slaves). The method employed there is rather crude and based on automatic naming conventions for service units: for every new service unit, the charm can extract its ordering number directly from its name (which is automatically assigned by the framework).

  Note: In fact the `mysql` charm is even more complex, because it also manages master-slave relations between separate deployed `mysql` services.

Relation

RELATION is a mechanism conceived for services to let them communicate with each other and exchange information about configuration parameters.

- Each relation has a *relation name* (an id) and an *interface name* (something more or less like a type).
- Services can react to various events concerning their relations thanks to the system of hooks defined by their charm.
- Exchange of information through a relation takes form of accessing and modifying so-called SETTINGS (key-value pairs), which are available to all services participating in the relation.

A little precision: relations are managed (defined, created, etc.) on the service level but they really work (are joined or departed, trigger hooks, etc.) on the service unit level. Therefore, a relation is defined globally for the whole service, but it is always realized by specific service units who join the relation and exchange information through it.

Charm

CHARM provides a definition of the service:

- Basic meta-data: name and description.
- Definition of possible or obligatory relations with other services:
  - requires / provides : client-server relations
  - peers : peer-to-peer relations
- Logic for management of the application: the hooks.

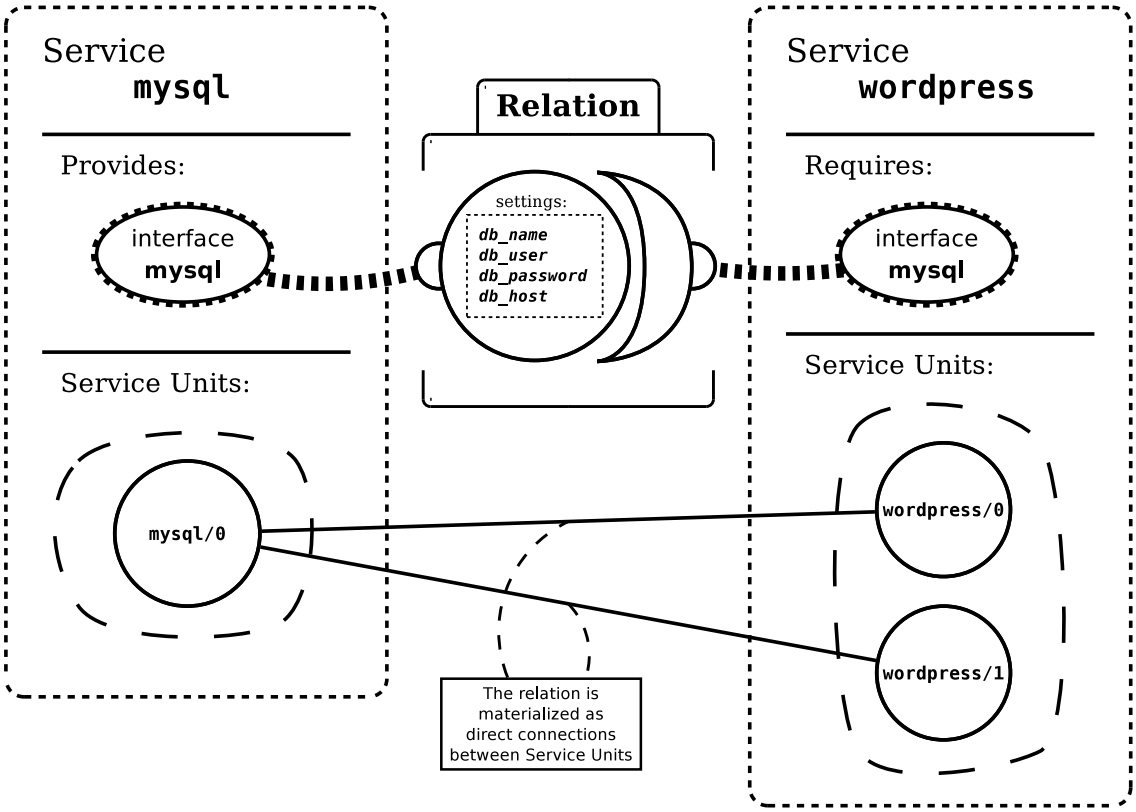


Figure A.4: Diagram illustrating components of a JUJU environment.

- Software packages on which the service depends.

  (Note: In fact using the software packages dependency information in any constructive way is not implemented yet. It is a future functionality. For now the task of installing all the required software packages is handled entirely by hand by hooks, so this meta-data field serves only for documentation purposes.)

Hook

HOOKS are in fact just simple event handlers: each of them is an executable file with its name exactly corresponding to the event they are meant to handle. When the event happens, the matching hook is triggered.

JUJU is using hooks to notify a service unit about changes happening in its life cycle and in the larger distributed environment (relations, service configuration, etc).

Hooks can be written in any programming language, the only restriction is that they have to be executable. Typically they are just bash scripts.

List of hooks and their corresponding events

This is a list of all the hooks that we can define for a charm, with indication in what circumstances they are triggered:

- `install` : service unit has been deployed
- `start` : service unit has been started
- `stop` : service unit has been stopped
- `X-relation-joined` : remote service unit joined the relation X
- `X-relation-changed` : remote service unit changed the relation X's settings
- `X-relation-departed` : remote service unit leaved the relation X
- `X-relation-broken` : relation X became unavailable
- `config-changed` : service configuration has been changed
- `upgrade-charm` : this charm has just been upgraded

Note: all hooks are optional!

Shell variables

In order to pass hooks some basic information about context of their execution, they are always invoked in a standard environment containing several special variables:

- `$JUJU_UNIT_NAME` : the name of the local service unit (the one executing the hook).
- `$JUJU_RELATION` : the name of the relation this hook is running for (only set if the hook was triggered by an event associated with a relation).
- `$JUJU_REMOTE_UNIT` : the name of the remote service unit, which has triggered the event, that caused the hook's execution (also only for relation-linked events).

Hook command tools

In hooks it is possible to access and modify certain aspects of JUJU environment using special command tools:

- `relation-get` command lets us access the information passed through the relation from a remote service unit.
- `relation-set` command lets us modify information, which we pass through the relation to a remote service unit.
- `relation-list` command lets us retrieve the list of all service units participating in this relation (useful for peer-to-peer relations).
- `config-get` command lets us access the service configuration.
- `unit-get` command lets us access some service unit-specific variables (like the public address of the machine on which we are).
- `open-port` and `close-port` commands let us control the firewall's settings.

A.5.3 User interface : juju commands

These are the most important commands, which let the user interact with JUJU (everything else lies in writing charms):

- `juju bootstrap` : bootstrap a new environment.
- `juju deploy` : deploy a charm as a service.
- `juju add-unit` : add a service unit to a service.
- `juju add-relation` : add a relation between two specified services (we can precise the relation name on both sides; if we do not, a relation with matching interfaces will be connected).
- `juju destroy-service`, `juju remove-unit`, `juju remove-relation` : remove a whole service, remove a single service unit, remove a relation.
- `juju expose`, `juju unexpose` : control access to a service from the internet (it just opens/closes the right port in the cloud's firewall).
- `juju upgrade-charm` : upgrade a service's charm

  (Note: this command has nothing to do with upgrading packages, just charms themselves!)

A.5.4 Internals

In this section we present some details about JUJU's interface, its internal structure and some of the mechanisms and solutions used.

Environment

Configuration JUJU can work simultaneously with multiple configured JUJU environments. Every JUJU environment has to be based on a single cloud environment. As we have mentioned before, for now the only supported Cloud Provider is *Amazon*. Therefore all of our JUJU environments must be based on *Amazon EC2* cloud environments. We can manage many separate instances of them at a time if we want. There is no communication between two separate environments on the JUJU level (that is more or less the point of having separate environments).

Metadata about all the configured environments, containing mainly the access and authorization information, is stored in a single file on the workstation: **environments.yaml**. Here we can see an example of configuration of a single *EC2* Environment called "sample":

```
environments:
  sample:
    type: ec2
    access-key: YOUR-ACCESS-KEY-GOES-HERE
    secret-key: YOUR-SECRET-KEY-GOES-HERE
    control-bucket: juju-faefb490d69a41f0a3616a4808e0766b
    admin-secret: 81a1e7429e6847c4941fda7591246594
```

Bootstrapping an environment In order to use a cloud environment as a configured JUJU environment, we always need to initialize it first – perform the BOOTSTRAPPING procedure. This operation is conducted by the JUJU client on our workstation and can be started by executing a simple command: `juju bootstrap`.

During the bootstrapping process, JUJU client (using the Cloud Provider) brings up the first machine in the environment. This utility instance, so-called BOOTSTRAPPING NODE, becomes the central point of the new environment and is used in all subsequent operations: it allows us to launch and orchestrate other instances.

Managing machines When the bootstrapping process is finished, all further cloud-specific operations are conducted via the bootstrapping node and JUJU AGENTS living inside the environment.

Therefore executing the `juju bootstrap` command is almost the only occasion, when the JUJU client on our workstation interacts with the Cloud Provider directly. After that moment, when the need of performing operations like managing the instances (provision a new instance / delete an existing instance) or changing settings of the cloud's firewall (open port / close port) arises, it is always one of JUJU's agents, living inside the cloud environment, who is responsible for performing all the necessary actions and handling the communication with the Cloud Provider.

Agents and the ZooKeeper

Agents AGENTS are a set of distributed processes living within the JUJU environment, tasked individually with various JUJU roles.

There are several types of agents:

- PROVISIONING AGENTS are responsible for allocating and terminating machines in the environment as necessary. As they are the only component in JUJU capable of interacting with the Cloud Provider, they also perform all the other operations concerning the cloud environment (like opening and closing ports in the cloud's firewall, etc). They live on the bootstrapping node.
- MACHINE AGENTS live on all the working machines. Every time when a new instance is provisioned, a machine agent is launched on it automatically. It is his duty to manage that single machine and handle all the requests concerning it (like a demand of deploying a charm on it).
- SERVICE UNIT AGENT (also known as UNIT AGENT or CHARM AGENT) is launched by the machine agent when a charm is deployed on his machine. He is responsible for managing the service unit, that has just been created, which basically means executing the hooks of the deployed charm in response to their corresponding events.
- RECOVERY AGENTS are responsible for monitoring the state of all running agents and restarting them when necessary.

ZooKeeper *Apache*'s *ZooKeeper* is a centralized service for maintaining configuration information. JUJU is using the virtual filesystem provided by the *ZooKeeper* to store its runtime state. RUNTIME STATE represents the complete current status of the JUJU's environment and serves as a centralized communication and synchronization mechanism for the agents.

The state stored in the *ZooKeeper*'s filesystem is fully introspectable and observable. The changes performed on it are atomic and globally ordered. These features let JUJU maintain its runtime state in a reliable and fault tolerant fashion, in the same time keeping it easily accessible and modifiable by all the concerned agents.

Communicating through the *ZooKeeper* Each agent process interacts with the runtime state (stored by the *ZooKeeper* centrally, on the bootstrapping node) and with the environment to perform its role.

In JUJU no messages are ever sent directly to a specific agent. Instead, a certain part of the runtime state gets modified and the agents watching this part respond to the change, usually by realizing the modifications in the environment. This method is used both when we want to send a command to an agent (i.e. order him to do something) and when two or more agents need to coordinate with each other.

Therefore, all the orchestration in JUJU is implemented on the bottom level with these two basic mechanisms:

- writing to the virtual filesystem,
- subscribing to certain parts of the virtual filesystem
  (which means watching them and reacting when they change).

ZooKeeper's filesystem structure The *ZooKeeper*'s virtual filesystem consists of nodes (so-called "znodes"). Each node can store data (in YAML format) and have sub-nodes. An agent can subscribe to certain node, he is then considered watching it and he is notified every time when this node changes.

Agents manifest themselves in the virtual filesystem using so-called EPHEMERAL NODES, which are dynamically created and removed to indicate the agent's presence. Presence of an agent in certain place means something like "I am here, I am alive and I am handling this now". For example, each machine's node in the virtual filesystem has a sub-node designated for its machine agent (and occupied by him if only he is running).

As ephemeral nodes are data structures in the same way that all the other nodes, they are also used by the agent processes simply to store their private data in a persistent fashion.

This is how the runtime state is represented by the hierarchy of *ZooKeeper*'s nodes:

```
/
├── topology
├── agents
├── charms
│   └── <namespace>:<name>-<revision> ..... one entry for each charm
├── services
│   └── service-<0..N> ..... one entry for each service
│       └── settings
├── units
│   └── unit-<0..N> ..... one entry for each service unit
│       ├── machine
│       └── charm-agent-connected
└── machines
    └── machine-<0..N> ..... one entry for each machine
        ├── provisioning-lock
        └── machine-agent-connected
```

ZooKeeper's filesystem explanation

/topology – describes the current topology of machines, services and service units.

/agents – each running agent has one entry in this node.

/charms – each charm used in the environment has one entry in this node:

/<namespace>:<name>-<revision> – contains metadata and settings concerning this charm.

/services – each deployed service (and a to-be-deployed service) has one entry in this node:

/service-<0..N> – contains the name of the deployed charm (corresponding exactly to a name of one of the nodes in /charms).

/settings – contains the service configuration.

/units – each entry in this znode corresponds to a running unit agent and therefore to a deployed service unit:

/unit-<0..N> – a single service unit.

/machine – the id of the machine on which this service unit is running.

/charm-agent-connected – ephemeral node for the unit agent.

/machines – each running machine (or a machine in the course of being provisioned) has one entry in this node:

/machine-<0..N> – a single machine.

/info – contains Cloud Provider launch information about the machine (*EC2* machine id, etc).

/provisioning-lock – a lock used by the provisioning agent.

/machine-agent-connected – ephemeral node for the machine agent.

juju in action

These are several typical scenarios, which take place in JUJU, described step-by-step with some details.

(This information has been extracted almost directly from JUJU documentation. Style was changed to improve understandability and some corrections based on other parts of documentation were introduced.)

Provisioning a new machine When the need for a new machine is determined, the following sequence of events happens in the JUJU environment and inside the *ZooKeeper* filesystem in order to deploy the new machine (lets say it is a machine number X):

1. A new node is created at **/machines/machine-X**
2. As the provisioning agent is watching the **/machines** node, he gets immediately notified about the new node.
3. The provisioning agent acquires a provisioning lock at **/machines/machine-X/provisioning-lock**
4. The provisioning agent checks if the machine still has to be provisioned by verifying if **/machines/machine-X/info** (containing Cloud Provider launch information) exists:
   (a) If this node does not exist, it means that the machine is currently not in the course of being provisioned. Therefore the provisioning agent fires a new instance via the Cloud Provider, stores the Cloud Provider launch information in **/machines/machine-X/info** and schedules to come back to the machine after `MachineBootstrapMaxTime` (a provisioning agent's setting).

(b) If the machine already has the Cloud Provider launch information, that means it is currently being provisioned, so the provisioning agent schedules to come back to the machine after `MachineBootstrapMaxTime`.

5. As a result of a schedule call, the provisioning agent verifies the existence of a **/machines/machine-X/machine-agent-connected** node, which is the ephemeral node of the machine agent running on our just provisioned machine:

   (a) If the node exists, it means that the machine agent is running, so the process of provisioning the machine has ended successfully. The provisioning agent sets a watch on this node, to be notified in case if the machine agent dies one day.

   (b) If the node does not exist after the `MachineBootstrapMaxTime`, then the provisioning agent supposes something went wrong during the provisioning process. He acquires the **/machines/machine-X/provisioning-lock**, terminates the instance, and goes back to step 4a (i.e. he tries to launch the machine again via the Cloud Provider)

6. In the meantime (what happens between step 4 and 5):
   When the new instance is provisioned by the Cloud Provider, it is automatically initialized: it installs all packages needed by the machine agent to run and it launches the machine agent itself. The first action, that the machine agent carries out, is connecting to the *ZooKeeper* filesystem and creating an ephemeral node for himself at **/machines/machine-X/machine-agent-connected**.

Deploying a Charm The orchestration of the process of deploying a charm to a machine (creating a service unit on that machine) is quite straightforward. In normal circumstances, as instances in the cloud are provisioned only when there is a demand for a new service unit, the charm deployment always happens directly after the machine has just been provisioned and initialized.

1. A new node is created as a sub-node of **/machines/machine-X/services**, to indicate that the machine X is supposed to be part of a certain service.

2. As the machine agent of the machine X is watching the **/machines/machine-X/services** node, he gets immediately notified about the new node.

3. The machine agent resolves which charm has to be deployed, downloads the charm (and all the files associated with it) to his machine and launches a unit agent for this charm.

4. From this moment, the new service unit is officially up and the recently created unit agent is entirely responsible for it. His job includes mainly one principal duty: he has to execute the hooks of the deployed charm when they should be executed. This entails interacting with the *ZooKeeper* filesystem: watching certain nodes (in order to react to events) and reading or writing to certain nodes (when a hook uses some of the special JUJU commands to access or modify variables like the service configuration or the settings passed through relations). For example, directly after being launched, the unit agent executes two hooks concerning the service unit's lifecycle: `install` and then `start`.

Changing Relations Handling changes in the relations between services is rather simple in the orchestration context. The unit agents responsible for the service units on all the machines are watching certain nodes in the *ZooKeeper* filesystem and react to their changes simply by launching right hooks locally on their machines.

Without entering into details about exactly which nodes are used to denote information concerning relations, the mechanism works like that:

- When a remote service unit joins the relation X, unit agents of all the concerned service units launch two hooks on their machines: first `X-relation-joined` and then `X-relation-changed`.

- When a remote service unit changes the settings of the relation X, the unit agents launch just one hook: `X-relation-changed`.
- And so on for other events concerning relations.

A.5.5 Limitations

JUJU's level of abstraction limits our scope to quite high-level concepts: we do not know anything about low-level resources like software packages or files. For JUJU in its current state this does not pose any problems. However, it may prove to be a major obstacle in implementing some of its important future functionalities (and a few of them are supposedly already in development!).

Currently JUJU can only deploy one service unit per machine. But if we wanted to deploy more service units on one machine, we would quickly find out that we do not have any means to recognize and resolve package (and package versions) conflicts nor configuration file conflicts. More generally, we cannot manage dependencies and conflicts concerning many important machine resources in any sensible way. The only option would be to work around the JUJU's model and manage it all manually on the low level, using hooks directly. Scripts, that we write, would have to be very careful and check everything thoroughly when editing configuration files, setting port numbers, etc. or they would have to communicate with each other through some unofficial way (like using specific files on the local file system).

Moreover, we do not have any way in JUJU to describe relations between two service units working on the same machine. It is worth noticing, that if we had, we would be (partially) able to solve the previous problem, because there would exist an official communication channel through which two service units deployed on the same machine could exchange configuration information.

Another potential source of trouble is the lack of mechanisms to orchestrate updates of software packages. If we wanted to perform updates in a proper fashion, we would quickly discover, that we have no idea, how services and software packages (and their different versions) work together and depend on each other. Therefore, we cannot reason about upgrades and ensure, that a given upgrade will be consistent and successful.

Hence, we can assume, that if JUJU's developers want to introduce all the features, which they have promised, they will probably need to alter their model soon. Or at least facilitate somehow the possibilities of working around the current model's shortcomings. An alternative option could be to withdraw from certain development ideas and sacrifice some planned functionalities (like the possibility to put multiple service units on one machine) in order to preserve the current simple spirit of JUJU.