Universidad Zaragoza

Proyecto Fin de Carrera
Ingeniería en Informática

Diseño e implementación de un sistema dinámico de gestión de trabajos distribuidos en un entorno de máquinas virtuales.

David Ceresuela Palomera

Director: Javier Celaya

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Curso 2011/2012
Septiembre 2012

Resumen

A la hora de ejecutar trabajos en un entorno distribuido, la aproximación clásica ha sido bien el uso de un *cluster* de ordenadores o bien el uso de la computación en malla o *grid*. Con la proliferación de entornos *cloud* durante estos últimos años y su facilidad de uso, una nueva opción se abre para la ejecución de este tipo de trabajos.

De hecho, la ejecución de trabajos distribuidos es uno de los principales usos dentro del ámbito de los sistemas *cloud*. Sin embargo, la administración de este tipo de sistemas dista de ser sencilla: cuestiones como la puesta en marcha del sistema, el aprovisionamiento de nodos, las modificaciones del sistema y la evolución y actualización del mismo suponen una tarea intensa y pesada.

En vista de lo cual, en este proyecto se ha diseñado una solución capaz de automatizar la administración de sistemas *cloud*, y en particular de un sistema de ejecución de trabajos distribuidos. Para ello se han estudiado entornos clásicos de ejecución de trabajos como TORQUE y entornos de ejecución de trabajos en *cloud* como AppScale. Además, se han estudiado herramientas clásicas de configuración automática de sistemas como Puppet y CFEngine. El objetivo principal de estas herramientas de configuración de sistemas es la gestión del nodo. En este proyecto se ha extendido la funcionalidad de una de estas herramientas — Puppet — añadiéndole la capacidad de gestión de sistemas *cloud*.

Como resultado de este proyecto se presenta una solución capaz de administrar de forma automática sistemas de ejecución de trabajos distribuidos. La validación de esta solución se ha llevado a cabo sobre los entornos de ejecución de trabajos TORQUE y AppScale y también, para mostrar su carácter genérico, sobre una arquitectura de servicios web de tres niveles.

Índice general

Capítulo 1
Introducción

[Revisar]

La computación en la nube es un nuevo paradigma que pretende transformar la computación en un servicio. Durante estos últimos años la computación en la nube ha ido ganando importancia de manera progresiva, ya que la posibilidad de usar la computación como un servicio permite a los usuarios de una aplicación acceder a ésta a través de un navegador web, una aplicación móvil o un cliente de escritorio, mientras que la lógica de la aplicación y los datos se encuentran en servidores situados en una localización remota. Esta facilidad de acceso a la aplicación sin necesitar de un profundo conocimento de la infraestructura es la que, por ejemplo, brinda a las empresas la posibilidad de ofrecer servicios web sin tener que hacer una gran inversión inicial en infraestructura propia. Las aplicaciones alojadas en la nube tratan de proporcionar al usuario el mismo servicio y rendimiento que las aplicaciones instaladas localmente en su ordenador.

A lo largo de los últimos años las herramientas de gestión de configuración (o herramientas de administración de sistemas) también han experimentado un considerable avance: con entornos cada vez más heterogéneos y complejos la administración de estos sistemas de forma manual ya no es una opción. Entre todo el conjunto de herramientas de gestión de configuración destacan de manera especial Puppet [1] y CFEngine. Puppet es una herramienta basada en un lenguaje declarativo: el usuario especifica qué estado debe alcanzarse y Puppet se encarga de hacerlo. CFEngine, también con un lenguaje declarativo, permite al usuario un control más detallado de cómo se hacen las cosas, mejorando el rendimiento a costa de perder abstracciones de más alto nivel.

Sin embargo, estas herramientas de gestión de la configuración carecen de la funcionalidad requerida para administrar infraestructuras distribuidas. Son capaces de asegurar que cada uno de los nodos se comporta de acuerdo a la configuración que le ha sido asignada pero no son capaces de administrar una infraestructura distribuida como una entidad propia. Si tomamos la administración de un *cloud* como la administración de las máquinas virtuales que forman los nodos del mismo nos damos cuenta de que la administración es puramente *software*. Únicamente tenemos que asegurarnos de que para cada nodo de la infraestructura distribuida hay una máquina virtual que está cumpliendo con su función.

Teniendo en cuenta el considerable avance de la computación en la nube, parece claro que el siguiente paso de las herramientas de gestión de configuración debería ir encaminado a la gestión de la nube. Para demostrar una posible manera en la que esto se podría lograr, en este proyecto se ha tomado una de esas herramientas de gestión de la configuración y se ha modificado añadiéndole la posibilidad de gestionar infraestructuras distribuidas. La modificación realizada se

ha validado usando tres ejemplos de infraestructuras distribuidas, que se explican a continuación.

La primera de ellas es AppScale [2], una implementación *open source* del App Engine de Google [7]. App Engine permite alojar aplicaciones web en la infraestructura que Google posee. Además del alojamiento de aplicaciones web, AppScale también ofrece las APIs [1] de EC2 [3], MapReduce [4] y Neptune [5]. La API de EC2 añade a las aplicaciones la capacidad de interactuar con máquinas alojadas en Amazon EC2 [6]. La API de MapReduce permite escribir aplicaciones que hagan uso del *framework* MapReduce. La última API, Neptune, añade a App Engine la capacidad de usar los nodos de la infraestructura para ejecutar trabajos. Los trabajos más representativos que puede ejecutar son: de entrada, de salida y MPI [2]. El trabajo de entrada sirve para subir ficheros (generalmente el código que se ejecutará) a la infraestructura, el de salida para traer ficheros (generalmente los resultados obtenidos después de la ejecución) y el de MPI para ejecutar un trabajo MPI.

La segunda infraestructura es TORQUE [13, 10], una infraestructura de ejecución de trabajos. Este tipo de infraestructuras está especializada en la ejecución de grandes cargas de trabajo paralelizable e intensivo en computación. Son por lo tanto idóneas para ser usadas en la computación de altas prestaciones.

La tercera y última es la de servicios web en tres capas. Este tipo de infraestructura tiene tres niveles claramente diferenciados: balanceo o distribución de carga, servidor web y base de datos. El balanceador de carga es el encargado de distribuir las peticiones web a los servidores web que se encuentran en el segundo nivel de la infraestructura. Éstos procesarán las peticiones web y para responder a los clientes puede que tengan que consultar o modificar ciertos datos. Los datos de la aplicación se encuentran en la base de datos, el tercer nivel de la estructura, y por consiguiente, cada vez que uno de los elementos del segundo nivel necesite leer información o modificarla, accederá a este nivel. Para esta infraestructura no se puede elegir un ejemplo que destaque sobre los demás porque es tan común que cualquier página web profesional de hoy en día se sustenta en una infraestructura similar a ésta.

1.1 Contexto del proyecto

Para la realización de este proyecto de fin de carrera se ha hecho uso del laboratorio 1.03b de investigación que el Departamento de Informática e Ingeniería de Sistemas posee en la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza. Los ordenadores que forman este laboratorio poseen procesadores con soporte de virtualización, lo que permite la creación de diversas máquinas virtuales. La creación de los distintos tipos de *cloud* que representan cada una de las infraestructuras distribuidas se ha llevado a cabo a través de máquinas virtuales alojadas en distintos ordenadores del laboratorio.

En este laboratorio se ha comprobado la validez de la extensión introducida en la herramienta de gestión de configuraciones Puppet para administración de infraestructuras distribuidas que se ha desarrollado a lo largo de este proyecto de fin de carrera.

[1] API (del inglés *Application Programming Interface*, Interfaz de programación de aplicaciones) es el conjunto de funciones y procedimientos que ofrece una biblioteca para ser utilizado por otro *software* como una capa de abstracción.

[2] MPI (del inglés *Message Passing Interface*, Interfaz de Paso de Mensajes) es un estándar que define la sintaxis y la semántica de las funciones de una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.

1.2 Objetivos

El objetivo de este proyecto es proporcionar una herramienta que facilite la puesta en marcha de infraestructuras distribuidas y su posterior mantenimiento. Las tareas principales en las que se puede dividir este proyecto son:

1. Análisis de las herramientas de administración de virtualización *hardware*.
2. Estudio de algunas de las infraestructuras distribuidas existentes profundizando en la parte relativa a la ejecución de trabajos distribuidos.
3. Investigación de las herramientas de gestión de configuración existentes más relevantes y elección de aquella que mayor facilidad de integración y uso proporcione.
4. Extensión de la herramienta de gestión de configuración para que soporte la puesta en marcha y el mantenimiento de un sistema de ejecución de trabajos distribuidos.

1.3 Trabajos previos

Desde un primer momento se decidió trabajar con la herramienta de configuración Puppet para la realización de este proyecto. La otra alternativa posible era CFEngine, pero a diferencia de ésta, Puppet posee un nivel mayor de abstracción que permite un mejor modelado de los recursos de un sistema. Además, el hecho de que Puppet esté programado en Ruby hace que sea más fácil trabajar y realizar abstracciones de alto nivel en él que en CFEngine, que está programado en el lenguaje C.

También se decidió desde el principio trabajar con la infraestructura de ejecución de trabajos que proporciona AppScale. AppScale combina la capacidad de ejecutar trabajos con el alojamiento de aplicaciones web. Esta dualidad la convierte en una infraestructura muy interesante para trabajar con ella.

La infraestructura de ejecución de trabajos TORQUE también se eligió desde el inicio.

1.4 Tecnologías utilizadas

Para la elaboración de este proyecto se ha hecho uso de las siguientes tecnologías: KVM, QEMU, libvirt y virsh para el soporte y la gestión de las máquinas virtuales; Puppet como herramienta de configuración automática; Ruby como lenguaje de programación para la extensión de Puppet; AppScale y TORQUE como infraestructuras de ejecución de trabajos distribuidos en las que validar la extensión; Nginx, WEBrick y MySQL como balanceador de carga, servidor web y base de datos para la infraestructura web de tres niveles en la que se valida la extensión; Shell como lenguaje de programación de los *scripts* de configuración de las máquinas virtuales; Sistema operativo Debian para las máquinas del laboratorio y Ubuntu para las máquinas virtuales; LaTeX [11] para la redacción de esta memoria y Dia para la elaboración de los diagramas que aparecen en esta memoria.

1.5 Herramientas utilizadas

[Revisar]

Una de las herramientas sobre las que se ha basado este proyecto ha sido la virtualización *hardware* o virtualización de plataforma, que permite la simulación de un ordenador completo

(llamado huésped) dentro de otro ordenador (llamado anfitrión). A la hora de hacer una virtualización *hardware* hay varias opciones entre las que elegir, siendo las más ampliamente usadas Xen y KVM. La principal diferencia entre ellas es que Xen ofrece paravirtualización mientras que KVM ofrece virtualización nativa.

La virtualización nativa permite hacer una virtualización *hardware* completa de manera eficiente. Para ello, y a diferencia de la paravirtualizacón, no requiere de ninguna modificación en el sistema operativo de la máquina virtual, pero a cambio necesita un procesador con soporte para virtualización. KVM, que proporciona virtualización *hardware*, está incluido como un módulo del núcleo de Linux desde su versión 2.6.20, así que viene incluido por defecto en cualquier sistema operativo con núcleo Linux.

Como los ordenadores del laboratorio poseen procesadores con extensiones de soporte para virtualización y sistema operativo Debian, se eligió KVM para dar soporte a las máquinas virtuales. Esto significa que se puede usar cualquier sistema operativo para las máquinas virtuales, sin necesidad de hacer ninguna modificación en el mismo.

El resto de herramientas utilizadas se explican en detalle en sus respectivas secciones.

1.6 Organización de la memoria

El resto de este documento queda organizado de la siguiente manera: el capítulo 2 describe las herramientas e infraestructuras utilizadas; el capítulo 3] indica cómo se ha realizado el modelado de recursos distribuidos con Puppet; el capítulo 4 introduce la metodología de diseño e implementación de recursos distribuidos; el capítulo 5 muestra como aplicar dicha metodología a la creación de recursos distribuidos; el capítulo 6 valida la solución planteada y el capítulo 7 presenta las conclusiones obtenidas e indica las posibilidades de trabajo futuro.

Además consta de una serie de anexos organizados de esta manera:

Anexo A Lenguaje declarativo de Puppet.

Anexo B Roles de AppScale.

1.7 Agradecimientos

Agradecimientos

Capítulo 2
Herramientas e infraestructuras utilizadas

[Revisar]

En este capítulo se realiza un breve análisis de las distintas herramientas e infraestructuras usadas a lo largo del proyecto.

2.1 Herramienta de gestión de configuración

Las herramientas de gestión de configuración tienen como objetivo describir y llevar a un sistema informático a un cierto estado. Normalmente esto suele incluir llevar a una serie de recursos (ficheros, usuarios, paquetes instalados, etc.) al estado deseado. Para cumplir esta misión se apoyan en dos conceptos básicos: iteración y convergencia. Estas herramientas tratan iteración tras iteración de acercar al sistema informático lo máximo posible al estado deseado. Es posible, por tanto, que el estado final no se alcance en una única ejecución, sino que sean necesarias varias ejecuciones.

Aunque esto pueda contrastar con el funcionamiento habitual de los programas (sería sorprendente que sólo se abriera medio editor de texto), no es tan excepcional en este entorno: si tenemos que poner en marcha dos servicios, de los cuales uno de ellos depende del otro, hasta que el primero no esté funcionando no podrá hacerlo el segundo. Una sola ejecución de la herramienta no serviría para poner ambos servicios en marcha, sino que serían necesarias dos iteraciones como mínimo.

Puppet [1] es una herramienta de gestión de configuración que posee un lenguaje declarativo (ver Anexo A) mediante el cual se especifica la configuración de los sistemas. A través de este lenguaje se especifica de forma declarativa los distintos elementos de configuración, que en la terminología de Puppet se llaman recursos. Mediante el uso de este lenguaje se indica en qué estado se quiere mantener el recurso y será tarea de Puppet el encargarse de que así sea. Cada recurso está compuesto de un tipo (el tipo de recurso que estamos gestionando), un título (el nombre del recurso) y una serie de atributos (los valores que especifican el estado del recurso).

A la especificación de forma declarativa de los recursos y del estado que estos deben alcanzar se le denomina manifiesto. En general, un manifiesto contiene la información necesaria para realizar la configuración de un nodo. Un administrador de un sistema informático que use Puppet creará manifiestos en los que especifique los recursos y su estado. Por ejemplo, para crear un fichero el administrador escribirá en un manifiesto algo similar a lo siguiente:

```
file {'testfile':
  path    => '/tmp/testfile',
  ensure  => present,
  mode    => 0640,
  content => "I'm a test file.",
}
```

Cuando se tiene listo el manifiesto a Puppet se le da la orden de aplicarlo. Los pasos que sigue para aplicarlo son:

- Interpretar y compilar la configuración.
- Comunicar la configuración compilada al nodo.
- Aplicar la configuración en el nodo.
- Enviar un informe con los resultados.

Normalmente Puppet se ejecuta de manera periódica mediante un planificador de trabajos (por ejemplo, cron). Cada cierto tiempo contactará con el nodo que debe ser administrado y volverá a repetir los pasos anteriores. Es decir, Puppet está continuamente intentando llevar al nodo al estado especificado en el manifiesto. Si entre una ejecución y otra algo cambiara en el nodo, Puppet se daría cuenta e intentaría llevar al nodo al estado que le corresponde.

Aunque Puppet tiene la capacidad de actuar sobre un nodo distinto al nodo desde el que se aplica el manifiesto, a lo largo de este proyecto las ejecuciones de Puppet siempre serán sobre el nodo que aplica el manifiesto, siempre serán locales.

2.2 Infraestructuras de ejecución de trabajos distribuidos

En esta sección se analizarán en profundidad las dos infraestructuras de ejecución de trabajos distribuidos usadas en este proyecto: AppScale y TORQUE

2.2.1 Análisis de la infraestructura AppScale

AppScale es una implementación *open source* del App Engine de Google. Al igual que App Engine, AppScale permite alojar aplicaciones web; a diferencia de App Engine, las aplicaciones no serán alojadas en la infraestructura que Google posee, sino que serán alojadas en una infraestructura que el usuario posea. Además de permitir alojar aplicaciones web, AppScale también ofrece las APIs de MapReduce y Neptune. La API de MapReduce permite escribir aplicaciones que hagan uso del *framework* MapReduce. La API de Neptune añade a App Engine la capacidad de usar los nodos de la infraestructura para ejecutar trabajos. Los trabajos más representativos que puede ejecutar son: de entrada, de salida y MPI, aunque también se pueden ejecutar trabajos de otro tipo.

AppScale trata de imitar de la manera más fiel los servicios que ofrece el App Engine de Google, tratatando de alcanzar el mayor grado de compatibilidad posible. Como la tecnología que Google usa para hacer esto posible permanece oculta al público, AppScale tiene que hacer uso de otras tecnologías existentes para ofrecer las mismas funciones. En la Figura 2.1 se puede ver toda la pila de tecnologías usadas en AppScale.

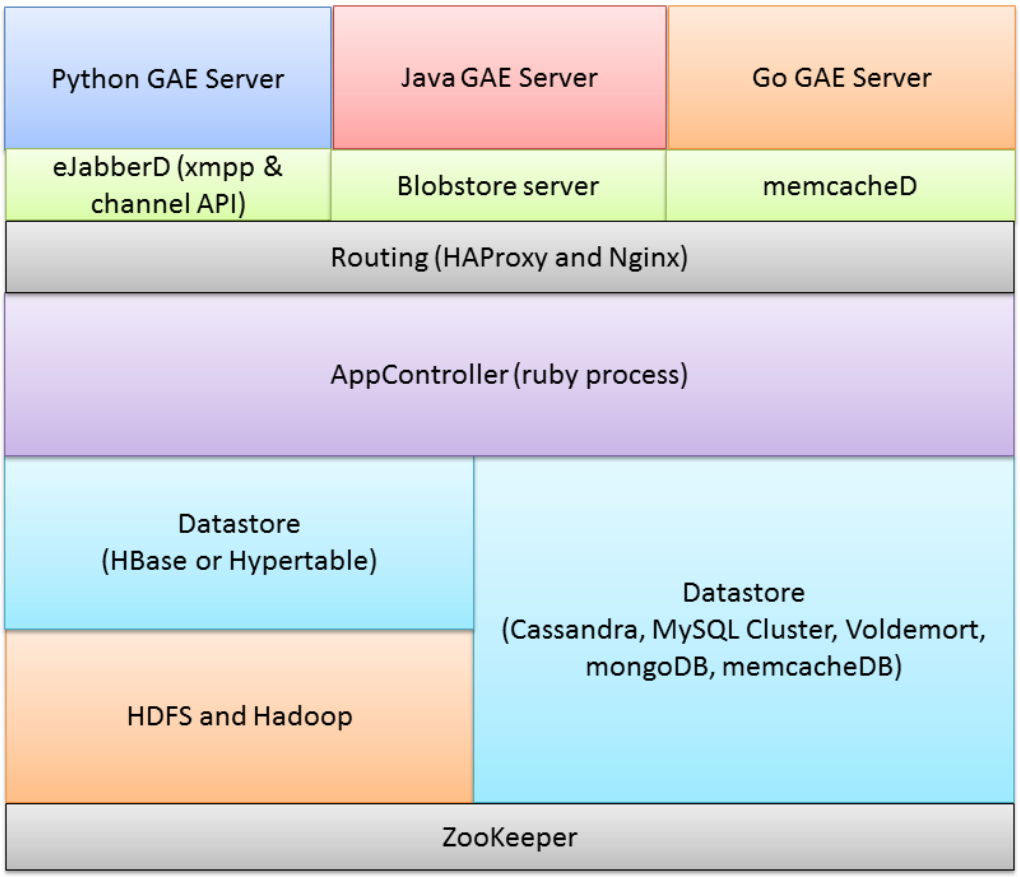

Figura 2.1: Tecnologías usadas por AppScale.

Para permitir el alojamiento de aplicaciones web y la ejecución de trabajos una compleja infraestructura debe ponerse en marcha. Los distintos roles que un nodo puede desempeñar en una infraestructura de este tipo son los siguientes:

`Shadow` : Comprueba el estado en el que se encuentran el resto de nodos.

`Load balancer` : Servicio web que lleva a los usuarios a las aplicaciones.

`AppEngine` : Aloja las aplicaciones web de los usuarios.

`Database` : Ejecuta los servicios necesarios para alojar a la base de datos elegida.

`Login` : Máquina desde la que se pueden hacer funciones administrativas.

`Memcache` : Proporciona soporte para almacenamiento en caché para las aplicaciones App Engine.

`Zookeeper` : Aloja los metadatos necesarios para hacer transacciones en las bases de datos.

`Open` : No ejecuta nada por defecto, pero está disponible en caso de que sea necesario.

Como muchos de estos roles suele desempeñarlos una máquina, se han creado unos roles agregados para facilitar el despliegue de la infraestructura. El rol `Controller` está compuesto por los roles Shadow, Load Balancer, Database, Login y Zookeeper; el rol `Servers` agrupa a los roles AppEngine, Database y Load Balancer; el rol `Master` está formado por los roles Shadow, Load Balancer y Zookeeper. Una especificación más detallada de los roles puede encontrarse en el Anexo B.

AppScale contempla la posibilidad de dos tipos de despliegue: por defecto y personalizado. En el despliegue por defecto un nodo sólo puede ser o bien `Controller` o bien `Servers`. En el despliegue personalizado el usuario es libre de especificar los roles de manera más precisa.

Una vez puesta en marcha la infraestructura de AppScale, servirá tanto para alojar las aplicaciones web que el usuario despliegue como para ejecutar trabajos. Para desplegar las aplicaciones web hay que hacer uso de las AppScale Tools, un conjunto de herramientas que permiten, entre otras cosas, iniciar y terminar instancias, desplegar aplicaciones y eliminar aplicaciones. Para ejecutar trabajos hay que servirse de la API de Neptune, que no es tan sencilla. En general esto se consigue mediante tres pasos: en el primero se le indica el código fuente que se quiere subir a la infraestructura; en el segundo se le da la orden de ejecutar el trabajo; en el tercero se le piden los resultados de la ejecución. Cada uno de estos pasos debe indicarse, mediante un lenguaje específico de dominio, en un fichero que luego se interpretará con el programa Neptune. En el caso de un trabajo MPI, podemos definir, además del código a ejecutar, el número de máquinas sobre las que ejecutar el código y el número de procesos que se usarán para el trabajo.

2.2.2 Análisis de la infraestructura TORQUE

La otra infraestructura de ejecución de trabajos distribuidos que se ha elegido ha sido TORQUE, una de las infraestructuras clásicas en lo que a ejecución de trabajos se refiere. TORQUE permite a los usuarios la ejecución de trabajos *batch* sobre un conjunto de nodos. Además de los nodos de computación necesarios para ejecutar los trabajos, una infraestructura de este tipo estará compuesta de un nodo maestro, que es el encargado de planificar la ejecución de los trabajos y de la gestión de los nodos de computación.

Una vez que la infraestructura está en funcionamiento los usuarios pueden mandar sus trabajos al nodo maestro. Éste, valiéndose de un planificador (en su versión más simple es una cola

FIFO), decidirá a cuál de los nodos de computación le enviará el trabajo. El nodo de computación que reciba el trabajo será el encargado de ejecutarlo y de enviar los resultados de vuelta al nodo maestro una vez terminada la ejecución.

Además de contar con un planificador básico, TORQUE permite la integración con otros planificadores tanto comerciales como *open source* para mejorar la planificación de trabajos y la administración de los nodos del *cluster*.

Capítulo 3
Modelado de recursos distribuidos con Puppet

[Revisar]

Las herramientas de gestión de la configuración se han centrado en la gestión de recursos de manera local a un nodo. Por otra parte la automatización existente en la administración de infraestructuras distribuidas es de bajo nivel, no yendo mucho más allá de la gestión de máquinas virtuales. En este capítulo veremos cómo se pueden crear recursos distribuidos en la herramienta de gestión de la configuración Puppet y cómo ésta puede ser usada para automatizar una administración de más alto nivel en infraestructuras distribuidas que tenga en cuenta conceptos como el de disponibilidad o el de prestaciones.

3.1 Configuración de recursos distribuidos

En Puppet, la definición clásica de recursos se presupone dentro del ámbito local del nodo. Es decir, para cada nodo especificamos qué recursos debe contener y cuál debe ser su estado. Dentro de este tipo de recursos se encuentran, entre otros, el recurso usuario y el recurso fichero. Sin embargo, el modelado de un recurso sistema distribuido plantea ciertos desafíos al ejemplo anterior, ya que no está pensado teniendo en cuenta la problemática asociada a los sistemas distribuidos.

Al modelar un recurso sistema distribuido, deben tenerse en cuenta las características propias de este tipo de recursos, como la disponibilidad, las prestaciones y las dependencias. La disponibilidad contempla los fallos que se pueden dar en una infraestructura distribuida y en ella estarían incluidos los fallos de procesos y los fallos de máquinas. Las prestaciones contemplan los servicios ofrecidos y dentro de ellas tendríamos la creación de máquinas para repartir la carga. Las dependencias contemplan la necesidad de que un servicio esté funcionando para que otro pueda hacerlo. Asimismo, un recurso sistema distribuido puede presentar elementos comunes con otros recursos sistema distribuido, tales como una monitorización básica. La presencia de elementos comunes entre los recursos clásicos de Puppet, por ejemplo un fichero y un usuario, no es tan corriente.

A la hora de definir un recurso sistema distribuido tenemos que presentarlo como un único sistema coherente, es decir, como una única abstracción, y por lo tanto no vale con describir un recurso sistema distribuido como una colección de recursos clásicos de Puppet. Afortunadamente, Puppet proporciona varios mecanismos de extensión. El primer mecanismo de extensión de Puppet consiste en la creación de nuevos tipos de recurso. El segundo es la creación de sub-

comandos y acciones dentro de Puppet mediante la API *Faces* [12]. Después de analizar esta API a fondo se vio que las opciones que ofrecía no permitían la integración del recurso sistema distribuido dentro del modelo de Puppet. Como lo que interesaba era crear una abstracción del recurso distribuido esta opción se acabó descartando en favor de la creación de un nuevo tipo de recurso y un proveedor asociado, que soluciona el problema de una manera más elegante: basta con crear un nuevo tipo de recurso con sus atributos correspondientes para definir los recursos sistema distribuido y basta con crear un proveedor que implemente los pasos necesarios para llevar dicho recurso al estado deseado.

3.2 Modelización en Puppet

Puppet puede ser extendido para incluir la definición de nuevos recursos. Para ello hay que proporcinarle como mínimo dos ficheros: uno en el que se define el recurso y otro en el que se define cómo gestionar ese recurso. Al fichero en el que se define el recurso se le llama tipo y al fichero en el que se define cómo gestionarlo se le llama proveedor. Es decir, el tipo se encarga del "qué" y el proveedor se encarga del "cómo".

Para definir un recurso sistema distribuido, al que también llamaremos *cloud*, se han considerado como fundamentales los siguientes atributos:

- Nombre: Para identificar al recurso de manera única.
- Fichero de dominio: Para definir una plantilla de creación de máquinas virtuales especificando sus características *hardware*.
- Conjunto de máquinas físicas: Para indicar qué máquinas físicas pueden ejecutar las máquinas virtuales definidas.

Estos atributos se obtienen mediante la observación de los elementos comunes a todo recurso sistema distribuido y forman, por tanto, el núcleo de un recurso *cloud* genérico. Además de estos atributos cada subtipo de recurso sistema distribuido (AppScale, TORQUE...) puede añadir los que considere necesarios para una completa especificación del recurso.

3.2.1 Patrón de diseño del proveedor

En Puppet, el proveedor es el encargado de llevar al recurso al estado que se le indique en el manifiesto. Típicamente el proveedor posee las funciones necesarias para crear un nuevo recurso y para destruirlo. Para llevar a cabo estas funciones en un recurso de tipo *cloud* el proveedor se apoya en tres grupos de funciones: puesta en marcha de un *cloud*, monitorización de un *cloud* y parada de un *cloud*. Las funciones de los dos primeros grupos se usan a la hora de crear un nuevo recurso de tipo *cloud* mientras que las del último grupo se usan a la hora de parar un *cloud* ya existente.

Puppet no contempla la herencia entre tipos o entre proveedores de distintos tipos. Por lo tanto, las funciones anteriormente especificadas no se encuentran dentro de ningún proveedor concreto (no existe, como tal, el proveedor de un recurso *cloud* genérico), sino que se presentan como una clase Ruby con unas funcionalidades básicas que los distintos proveedores de recursos sistema distribuido pueden usar para facilitar el desarrollo de los mismos.

Las operaciones de puesta en marcha son las encargadas de poner en funcionamiento el *cloud* especificado en el manifiesto. Las más importantes son:

- Inicio como líder: Función de puesta en marcha que realizará el nodo líder del *cloud*. Ésta es la función más importante dentro de las funciones de inicio del proveedor ya que es la que se encarga de iniciar el *cloud*. A grandes rasgos, los pasos que realiza son:

  1. Comprobación de la existencia del *cloud*: si no existe se creará.
  2. Comprobación del estado del conjunto de máquinas físicas.
  3. Obtención de las direcciones IP de los nodos y los roles que les han sido asignados.
  4. Comprobación del estado de las máquinas virtuales: si están funcionando se monitorizan, mientras que si no están funcionando hay que definir una nueva máquina virtual y ponerla en funcionamiento. Las funciones de monitorización incluyen el envío de un fichero mediante el cual cada nodo se autoadministre la mayor parte posible.
  5. Cuando todas las máquinas virtuales estén funcionando se procede a inicializar el *cloud*.
  6. Operaciones de puesta en marcha particulares dependiendo de cada tipo de *cloud*.

- Inicio como nodo común: Función de puesta en marcha que realizarán los nodos comunes del *cloud*.

- Inicio como nodo externo: Función de puesta en marcha que realizará un nodo no perteneciente al *cloud*.

La monitorización del *cloud* únicamente la llevará a cabo uno de los nodos, al que llamaremos líder, ya que sería redundate que más de un nodo se encargara de comprobar el estado global del *cloud*. Por tanto, sólo hay una función importante:

- Monitorización como líder: Función de monitorización que realizará el nodo líder del *cloud*.

Para elegir al nodo líder de entre todos los nodos del *cloud* se utiliza el algoritmo peleón (en inglés *Bully algorithm*). En este algoritmo todos los nodos tratan periódicamente de convertirse en el líder; si hay un líder, impedirá que otro nodo le quite el liderazgo, y si no lo hay, uno de los restantes nodos se convertirá en líder. Para ayudar a la implementación de este algoritmo se proporcionan las funciones de elección de líder, de las cuales las más importantes son:

- Lectura y escritura de identificador: Funciones de lectura y escritura del identificador del nodo actual.

- Lectura y escritura de identificador de líder: Funciones de lectura y escritura del identificador del nodo líder.

- Escritura de identificador e identificador de líder remoto: Funciones de escritura del identificador y del identificador del líder en un nodo distinto del actual.

Por último, es posible que en algún momento se desee parar por completo el funcionamiento del *cloud*. Las operaciones de parada de *cloud* más importantes son:

- Apagado de máquinas virtuales: Función de apagado de las máquinas virtuales que forman el *cloud*.

- Borrado de ficheros: Función de eliminación de todos los ficheros internos de gestión del *cloud*.

Aunque no se proporcionan como funciones, hay que tener en cuenta que cada tipo de *cloud* puede tener sus propias funciones de parada. Estas funciones de parada deben realizarse antes de apagar las máquinas virtuales. De forma general, los pasos que hay que hacer a la hora de parar un *cloud* son: [Revisar este orden]

1. Comprobación de la existencia del *cloud*: si existe se procederá a su parada.
2. Operaciones de parada particulares a cada tipo de *cloud*.
3. Apagado de las máquinas virtuales creadas explícitamente para este *cloud*.
4. Parada de las funciones de automantenimiento de los nodos.
5. Eliminación de los ficheros internos de gestión del *cloud*.

3.2.2 *Framework* de implementación

La mejor manera de crear un tipo y un proveedor en Puppet es hacerlo usando el concepto de módulo que ya dispone Puppet. Los módulos deben organizarse siguiendo una estructura predefinida. En esta estructura hay que colocar el código del tipo y del proveedor en la parte correspondiente. Si se crea el tipo `tipo` y el proveedor `proveedor` dentro del módulo `modulo` la estructura sería similar a la siguiente:

```
modulo/
modulo/manifests
modulo/manifests/init.pp
modulo/files
modulo/templates
modulo/lib/facter
modulo/lib/puppet/type
modulo/lib/puppet/type/tipo.rb
modulo/lib/puppet/provider
modulo/lib/puppet/provider/tipo/proveedor.rb
modulo/lib/puppet/parser/functions
```

Al ser el recurso sistema distribuido un recurso genérico, dicha organización es excesiva, puesto que carece de la mayoría de los elementos ahí definidos. Bastará por tanto con la siguiente estructura:

```
generic-cloud/provider
generic-cloud/provider/cloud.rb
```

Dentro del fichero `cloud.rb` se encuentra definida la clase `Cloud` que proporciona las funciones más básicas que ayudan al resto de recursos sistema distribuido a crear sus proveedores. En concreto, las operaciones de puesta en marcha que se proporcionan son `leader_start`, `common_start` y `not_cloud_start`; la función de monitorización que se proporciona es `leader_monitoring`; las funciones de elección de líder que se proporcionan son `get_id`, `set_id`, `get_leader`, `set_leader`, `vm_set_id` y `vm_set_leader`; por último, las funciones de parada de *cloud* que se proporcionan son `shutdown_vms` y `delete_files`. La implementación completa de la clase `Cloud` puede verse en el anexo X.

Además de proporcionar estas funciones, la clase `Cloud` necesita de otras para realizar su trabajo. En particular, cada proveedor debe implementar dos funciones. La primera de ellas es la

función `start_cloud` que se usará para iniciar el *cloud*. La segunda es la función `obtain_vm_data` que se utilizará para obtener datos de las máquinas virtuales que forman el *cloud*.

Para monitorizar el *cloud* concreto, también será necesario pasarle una función como argumento a las funciones `leader_start` y `leader_monitoring`. Esta función implementa una monitorización que incluye los aspectos particulares de cada tipo de *cloud*.

Capítulo 4

Metodología de diseño e implementación de recursos distribuidos

[Introducción]

4.1 Especificación del tipo

El primer paso que hay que hacer a la hora de crear un nuevo recurso en Puppet es definir el tipo de recurso. Para ello en un fichero habrá que especificar el nombre del nuevo tipo y los argumentos que éste tiene. Durante esta sección y la siguiente se usará el ejemplo de la arquitectura web de tres niveles ya que es la más conocida y sirve para demostrar que el enfoque que se ha tomado es lo suficientemente general y válido también para infraestructuras que nada tienen que ver con la ejecución de trabajos.

Una típica arquitectura de servicios web consta de al menos tres niveles: balanceo de carga, servidores web y base de datos. Cada uno de estos niveles está compuesto por al menos un elemento clave: el balanceador de carga, el servidor web y el servidor de base de datos, respectivamente. El balanceador de carga es el punto de entrada al sistema y el que se encarga, como su nombre indica, de repartir las peticiones de los clientes a los distintos servidores web. Los servidores web se encargan de servir las páginas web a los clientes y para ello, dependiendo de las peticiones que hagan los clientes, podrán leer o almacenar información en la base de datos. Para manipular dicha información los servidores web tendrán que comunicarse con el servidor de base de datos, que es el que hará efectiva la lectura y modificación de la información.

La infraestructura usada como ejemplo consta de un balanceador de carga, dos servidores web y un servidor de bases de datos (Figura 4.1).

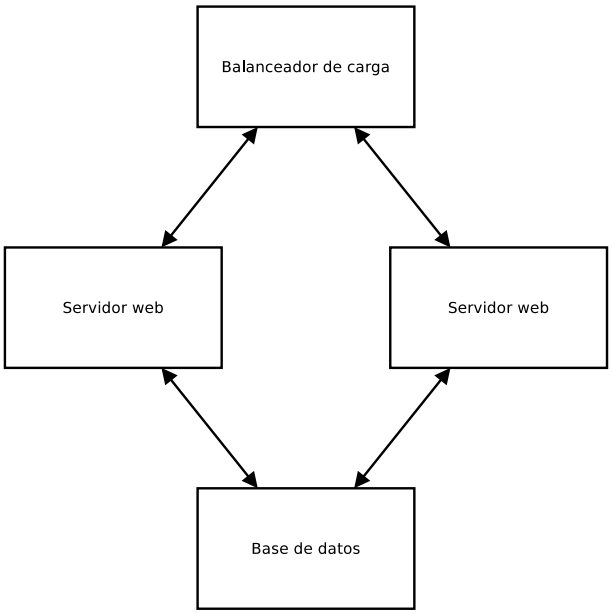

Figura 4.1: Infraestructura web de tres niveles.

La creación del tipo `web` se hace en el fichero `web.rb`, en el lugar apropiado dentro del correspondiente módulo. En el siguiente fragmento de código se indican los aspectos más importantes (la implementación completa puede encontrarse en el Anexo X):

```
1  Puppet::Type.newtype(:web) do
2     @doc = "Manages web clouds formed by KVM virtual machines."
3
4
5     ensurable do
6
7        desc "The cloud's ensure field can assume one of the following values:
8     'running': The cloud is running.
9     'stopped': The cloud is stopped.\n"
10
11       newvalue(:stopped) do
12          provider.stop
13       end
14
15       newvalue(:running) do
16          provider.start
17       end
18
19    end
20
21
22    # General parameters
23
24    newparam(:name) do
25       desc "The cloud name"
26       isnamevar
```

```
27      end
28
29      newparam(:vm_domain) do
30          desc "The XML file with the virtual machine domain definition. " +
31               "Libvirt XML format must be used"
32      end
33
34      newproperty(:pool, :array_matching => :all) do
35          desc "The pool of physical machines"
36      end
37
38      # ...
39
40
41      # Web parameters
42
43      newproperty(:balancer, :array_matching => :all) do
44          desc "The balancer node's information"
45      end
46
47      newproperty(:server, :array_matching => :all) do
48          desc "The server nodes' information"
49      end
50
51      newproperty(:database, :array_matching => :all) do
52          desc "The database node's information"
53      end
54
55  end
```

Dentro del tipo `web`, los atributos `name`, `vm_domain` y `pool` se corresponden con los atributos `Nombre`, `Fichero de dominio` y `Conjunto de máquinas físicas` definidos en la sección 3.2. Los atributos `balancer`, `server` y `database` son los atributos propios de la arquitectura web.

Una vez que el tipo ya está definido, ya se puede ver cuál será el aspecto de un manifiesto para un recurso de tipo web:

```
web {'mycloud':
    balancer => ["155.210.155.175", "/var/tmp/dceresuela/lucid-lb.img"],
    server => ["/etc/puppet/modules/web/files/server-ips.txt",
               "/etc/puppet/modules/web/files/server-imgs.txt"],
    database => ["155.210.155.177", "/var/tmp/dceresuela/lucid-db.img"],
    vm_domain => "/etc/puppet/modules/web/files/mycloud-template.xml",
    pool => ["155.210.155.70"],
    ensure => running,
}
```

4.2 Diseño e implementación del proveedor

Una vez que se ha definido el tipo del recurso, queda definir el proveedor que implementa dicho tipo. En el caso de los recursos distribuidos, y teniendo en cuenta las funciones de arranque y parada especificadas en el tipo (en el bloque `ensurable`, líneas 5 a 19), el proveedor deberá contener obligatoriamente las funciones `start` y `stop`. A continuación se presenta un proveedor con los aspectos más importantes (la implementación completa se encuentra en el Anexo X):

```
1  Puppet::Type.type(:web).provide(:webp) do
2     desc "Manages web clouds formed by KVM virtual machines"
3
4     # Require files
5     # ...
6
7     # Start function. Makes sure the cloud is running.
8     def start
9
10       cloud = Cloud.new(CloudInfrastructure.new(), CloudLeader.new(), resource,
11                         method(:err))
12       puts "Starting cloud %s" % [resource[:name]]
13
14       # Check existence
15       if !exists?
16          # Cloud does not exist => Startup operations
17
18          # Check pool of physical machines
19          puts "Checking pool of physical machines..."
20          pm_up, pm_down = cloud.check_pool()
21          unless pm_down.empty?
22             puts "Some physical machines are down"
23             pm_down.each do |pm|
24                puts " - #{pm}"
25             end
26          end
27
28          # Obtain the virtual machines' IPs
29          puts "Obtaining the virtual machines' IPs..."
30          vm_ips, vm_ip_roles, vm_img_roles = obtain_vm_data(cloud.resource)
31
32          # Check whether you are one of the virtual machines
33          puts "Checking whether this machine is part of the cloud..."
34          part_of_cloud = vm_ips.include?(MY_IP)
35          if part_of_cloud
36             puts "#{MY_IP} is part of the cloud"
37
38             # Check if you are the leader
39             if cloud.leader?()
40                cloud.leader_start("web", vm_ips, vm_ip_roles, vm_img_roles,
41                                   pm_up, method(:web_monitor))
42             else
```

```
43                cloud.common_start()
44             end
45          else
46             puts "#{MY_IP} is not part of the cloud"
47             cloud.not_cloud_start("web", vm_ips, vm_ip_roles, vm_img_roles,
48                                   pm_up)
49          end
50
51       else
52
53          # Cloud exists => Monitoring operations
54          puts "Cloud already started"
55
56          # Check if you are the leader
57          if cloud.leader?()
58             cloud.leader_monitoring(method(:web_monitor))
59          else
60             puts "#{MY_IP} is not the leader"      # Nothing to do
61          end
62       end
63
64    end
65
66
67    # Stop function. Makes sure the cloud is not running.
68    def stop
69
70       # ...
71
72    end
73
74
75    def status
76       if File.exists?("/tmp/cloud-#{resource[:name]}")
77          return :running
78       else
79          return :stopped
80       end
81    end
82
83 end
```

Son especialmente importantes dentro de este proveedor las líneas 10, 30 y 40. En la línea 10 creamos un nuevo objeto de la clase `Cloud` (3.2.2). En la línea 30 se llama a la función `obtain_vm_data` para obtener los datos de las máquinas virtuales (3.2.2). En la línea 40 puede verse como el objeto de la clase `Cloud` se usa para llamar al método `leader_start` que realizará las funciones de puesta en marcha como nodo líder del *cloud* (3.2.1). Uno de los argumentos del método en esa llamada es la función `web_monitor` que se usará para monitorizar los aspectos particulares del *cloud* web (3.2.2).

Merece la pena comentar que tanto el tipo como el proveedor están implementados haciendo uso de la capacidad de metaprogramación que ofrece Ruby. Ambos se pasan como un bloque de código Ruby a una función de Puppet (`newtype` y `provide` respectivamente) que se encargará internamente de realizar las operaciones necesarias. Es por este motivo que en el código del tipo y del proveedor aparecen variables (como `resource` y `provider`) y funciones (como `newparam` y `desc`) que no pertenecen al lenguaje Ruby y no se encuentran *a priori* definidas en ninguna parte. Gracias a la ejecución dinámica de Ruby sabemos que estas variables y funciones estarán definidas en el entorno de Puppet en tiempo de ejecución.

Capítulo 5
Diseño e implementación de recursos distribuidos

En este capítulo se verá cómo aplicar la metodología anteriormente expuesta para diseñar e implentar los recursos distribuidos de las infraestructuras de ejecución de trabajos AppScale y TORQUE.

5.1 Diseño e implementación de un recurso distribuido para una infraestructura AppScale

Una infraestructura AppScale puede ser definida de dos maneras: mediante un despliegue por defecto o uno personalizado. En un despliegue por defecto un nodo es el encargado de controlar la infraestructura y el resto de nodos se encargan de hacer el resto del trabajo. En un despliegue personalizado podemos especificar con mayor grado de precisión qué tipo de trabajo debe hacer cada nodo. Por ejemplo, podemos indicar qué nodos se encargarán de alojar las aplicaciones de los usuarios, qué nodos alojarán la base de datos o qué nodos serán los encargados de ejecutar los trabajos de computación. Para administrar una infraestructura AppScale, sin importar el tipo de despliegue, necesitaremos una cuenta de correo y una contraseña. Este usuario y contraseña son necesarios para poder administrar las aplicaciones alojadas y observar el estado de la infraestructura.

En un despliegue por defecto los posibles roles que puede tomar un nodo son:

`controller` : La máquina que desempeñará el rol de nodo controlador.

`servers` : La lista de máquinas que desempeñarán el rol de nodos de trabajo.

Mientras que los posibles roles que puede desempeñar un nodo en un despliegue personalizado y que resultan interesantes desde nuestro punto de vista son:

`master` : La máquina que desempeñará el rol de nodo maestro.

`appengine` : Los servidores para alojar las aplicaciones.

`database` : Las máquinas que contienen la base de datos.

`login` : La máquina encargada de redirigir a los usuarios a sus servidores. Es también la que se le facilita al administrador de la infraestructura para que realice las tareas administrativas.

`open` : Las máquinas de ejecución de trabajos. También pueden ser usadas como nodos de reserva por si falla algún otro nodo.

Hay multitud de despliegues posibles combinando estos roles, pero será de especial interés para este proyecto el que permite ejecutar trabajos de computación en AppScale (Figura 5.1).

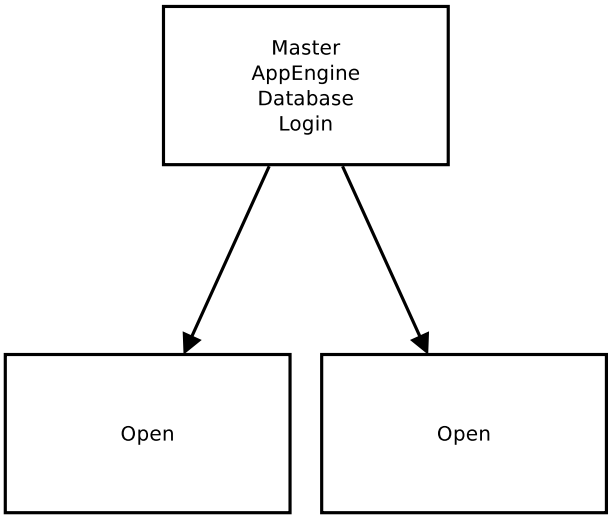

Figura 5.1: Infraestructura AppScale en despliegue personalizado.

Además de los atributos necesarios para los roles de las máquinas, el tipo `appscale` también debe contar con la cuenta de correo y la contraseña para el administrador del *cloud*. Los atributos más importantes del tipo `appscale` (ver Anexo X para implementación completa), en lo que a ejecución de trabajos concierne, son:

```
1  Puppet::Type.newtype(:appscale) do
2     @doc = "Manages AppScale clouds formed by KVM virtual machines."
3
4     # ...
5
6     # AppScale parameters
7
8     newproperty(:master, :array_matching => :all) do
9        desc "The master node"
10    end
11
12    newproperty(:appengine, :array_matching => :all) do
13       desc "The appengine nodes"
14    end
15
16    newproperty(:database, :array_matching => :all) do
17       desc "The database nodes"
18    end
19
20    newproperty(:login, :array_matching => :all) do
21       desc "The login node"
22    end
```

```
23
24      newproperty(:open, :array_matching => :all) do
25          desc "The open nodes"
26      end
27
28
29      newparam(:app_email) do
30          desc "AppScale administrator e-mail"
31          defaultto "david@gmail.com"
32      end
33
34      newparam(:app_password) do
35          desc "AppScale administrator password"
36          defaultto "appscale"
37      end
38
39  end
```

Para lograr un despliegue de este tipo podría utilizarse un manifiesto similar a éste:

```
appscale {'mycloud':
    master    => ["155.210.155.73",
                  "/var/tmp/dceresuela/lucid-tor1.img"],
    appengine => ["155.210.155.73",
                  "/var/tmp/dceresuela/lucid-tor1.img"],
    database  => ["155.210.155.73",
                  "/var/tmp/dceresuela/lucid-tor1.img"],
    login     => ["155.210.155.73",
                  "/var/tmp/dceresuela/lucid-tor1.img"],
    open      => ["/etc/puppet/modules/appscale/files/open-ips.txt",
                  "/etc/puppet/modules/appscale/files/open-imgs.txt"],
    vm_domain => "/etc/puppet/modules/appscale/files/mycloud-template.xml
        ",
    pool => ["155.210.155.70"],
    ensure => running,
}
```

5.2 Diseño e implementación de un recurso distribuido para una infraestructura TORQUE

Una infraestructura TORQUE está formada por un nodo maestro y un conjunto de nodos de computación (Figura 5.2). El nodo maestro es el encargado de recibir los trabajos a ejecutar y de asegurar una correcta planificación para esos trabajos; en su versión más simple el planificador es una cola FIFO. Los nodos de computación son los encargados de ejecutar los trabajos enviados por el nodo maestro y, una vez terminados, enviarle los resultados de vuelta.

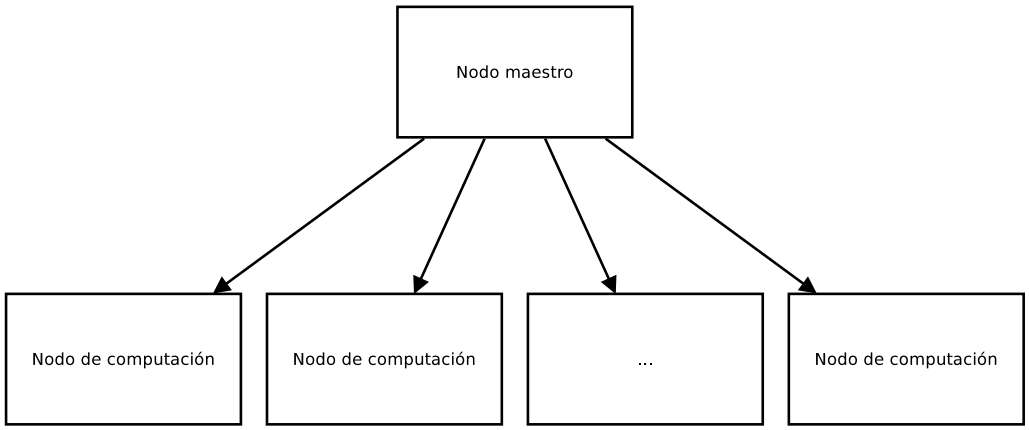

Figura 5.2: Infraestructura Torque.

Los nuevos atributos que aparecen en el tipo `torque` reflejan esta nueva infraestructura:

```
1  Puppet::Type.newtype(:torque) do
2     @doc = "Manages Torque clouds formed by KVM virtual machines."
3
4     # ...
5
6     # Torque parameters
7     newproperty(:head, :array_matching => :all) do
8        desc "The head node's information"
9     end
10
11    newproperty(:compute, :array_matching => :all) do
12       desc "The compute nodes' information"
13    end
14
15 end
```

La sintaxis del manifiesto de puesta en marcha también se modifica para reflejar este cambio. Un ejemplo de un manifiesto para la puesta en marcha de una infraestructura TORQUE sería similar a éste:

```
torque {'mycloud':
   head => ["155.210.155.73", "/var/tmp/dceresuela/lucid-tor1.img"],
   compute => ["/etc/puppet/modules/torque/files/compute-ips.txt",
               "/etc/puppet/modules/torque/files/compute-imgs.txt"],
   vm_domain => "/etc/puppet/modules/torque/files/mycloud-template.xml",
   pool => ["155.210.155.70"],
   ensure => running,
}
```

Capítulo 6
Validación de la solución planteada

x Máquinas físicas
y Máquinas virtuales
AppScale: 1 maestro, dos esclavos

Para validar la solución desarrollada, se ha hecho uso del laboratorio 1.03b que el Departamento de Informática e Ingeniería de Sistemas posee en la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza. Los ordenadores de este laboratorio poseen procesadores con soporte para virtualización, lo que hace posible la creación de máquinas virtuales para simular los nodos que forman cada una de las infraestructuras distribuidas.

Antes de empezar con las pruebas hay que configurar el entorno en el que se realizarán. En particular, y dado que las máquinas virtuales necesitan conectarse a internet para la descarga e instalación de paquetes, el uso de un servidor de DNS es bastante recomendable. De este modo, podemos usar direcciones IP públicas (que no se estén usando en ese momento) para nuestras máquinas virtuales. El servidor DNS se usa también para hacer la resolución de nombres, tanto normal como inversa, que requieren AppScale y TORQUE para su correcto funcionamiento.

6.1 Pruebas comunes a todas las infraestructuras

En todas y cada una de las infraestructuras se han realizado las siguientes pruebas para comprobar el correcto funcionamiento del proveedor distribuido:

- Apagado de una máquina virtual que había empezado encendida y no era líder.
- Apagado de una máquina virtual que no había empezado encendida y no era líder.
- Apagado de una máquina virtual que había empezado encendida y era líder.
- Puesta en marcha de la infraestructura desde una máquina que no pertenece al *cloud*.
- Puesta en marcha de la infraestructura desde una máquina que pertenece al *cloud*.

6.2 Prueba de infraestructura AppScale

6.3 Prueba de infraestructura TORQUE

Para probar la infraestructura TORQUE se han usado cuatro (de momento) máquinas virtuales alojadas en X máquinas físicas. Una de las máquinas virtuales actúa como nodo maestro y

las otras tres (de momento) actúan como nodos de computación. Además de las pruebas comunes, para comprobar el provedor de la infraestructura TORQUE se han realizado las siguientes pruebas:

- Parada del proceso de autenticación (trqauthd) en el nodo maestro.
- Parada del proceso servidor (pbs_server) en el nodo maestro.
- Parada del proceso planificador (pbs_sched) en el nodo maestro.
- Parada del proceso de ejecución de trabajos (pbs_mom) en un nodo de computación.
- Parada del proceso que monitoriza al proceso de autenticación en el nodo maestro.
- Parada del proceso que monitoriza al proceso servidor en el nodo maestro.
- Parada del proceso que monitoriza al proceso planificador en el nodo maestro.
- Parada del proceso que monitoriza al proceso de ejecución de trabajos en un nodo de computación.

6.4 Prueba de infraestructura web de tres niveles

Para probar la infraestructura web se han usado cuatro máquinas virtuales repartidas entre X máquinas físicas. Una máquina virtual actúa como balanceador de carga, dos actúan como servidores web y la última actúa como base de datos. Las pruebas que se han realizado para comprobar el correcto funcinamiento del proveedor de la infraestructura web han sido:

- Parada del proceso balanceador de carga.
- Parada del proceso servidor web.
- Parada del proceso base de datos.
- Parada del proceso que monitoriza al proceso balanceador de carga.
- Parada del proceso que monitoriza al proceso servidor web.
- Parada del proceso que monitoriza al base de datos.

Capítulo 7
Conclusiones

En este proyecto fin de carrera se ha creado un modelo de recursos distribuidos para facilitar la puesta en marcha y la automatización de infraestructuras distribuidas. Para comprobar la validez del modelo se ha extendido una herramienta de gestión de la configuración a la que se le ha añadido el recurso sistema distribuido o *cloud*. Posteriormente se ha comprobado la extensión realizada haciendo que la herramienta de gestión de configuración sea capaz de administrar infraestructuras distribuidas de ejecución de trabajos como AppScale y TORQUE. Además se ha visto que el modelo también es válido para infraestructuras más generales como la de servicios web en tres niveles.

Bibliografía

[1] Puppet labs: The leading open source data center automation solution. `http://www.puppetlabs.com/`.

[2] AppScale: An open-source implementation of the Google AppEngine (GAE) cloud computing interface. `http://appscale.cs.ucsb.edu/`, 2011.

[3] AppScale: EC2 API Documentation. `http://code.google.com/p/appscale/wiki/EC2_API_Documentation`, 2011.

[4] AppScale: MapReduce API Documentation. `http://code.google.com/p/appscale/wiki/MapReduce_API_Documentation`, 2011.

[5] AppScale: Neptune API Documentation. `http://www.neptune-lang.org/`, 2011.

[6] Amazon: Elastic Compute Cloud. `http://aws.amazon.com/ec2/`, 2012.

[7] Google: App Engine. `https://developers.google.com/appengine/`, 2012.

[8] Chris Bunch, Navraj Chohan, Chandra Krintz, and Khawaja Shams. Neptune: a domain specific language for deploying hpc software on cloud platforms. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, ScienceCloud '11, pages 59–68, New York, NY, USA, 2011. ACM.

[9] Navraj Chohan, Chris Bunch, Sydney Pang, Chandra Krintz, Nagy Mostafa, Sunil Soman, and Richard Wolski. Appscale: Scalable and open appengine application development and deployment. In *CloudComp*, pages 57–70, 2009.

[10] Adaptative Computing. TORQUE Resource Manager. `http://www.adaptivecomputing.com/products/open-source/torque/`, 2012.

[11] LaTeX project team. *LaTeX documentation*. `http://www.latex-project.org/guides/`.

[12] Puppet Labs. Puppet Faces API. `http://puppetlabs.com/faces/`, 2012.

[13] Garrick Staples. TORQUE resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[14] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby. The Pragmatic Programmer's Guide*. Pragmatic Programmers, 2004.

[15] J. Turnbull and J. McCune. *Pro Puppet*. Pro to Expert Series. Apress, 2011.

Apéndice A
Puppet

Puppet es una herramienta de gestión de configuración basada en un lenguaje declarativo. Este lenguaje declarativo permite expresar de una manera clara la configuración deseada en un sistema. Para expresar dicha configuración un usuario indicará en un fichero, llamado manifiesto, cuál es el estado que deben alcanzar los recursos de ese sistema. Si, por ejemplo, un usario quisiera especificar cuál debe ser el contenido de un fichero, podría utilizar algo similar a lo siguiente:

```
file {'testfile':
  path    => '/tmp/testfile',
  ensure  => present,
  mode    => 0640,
  content => "I'm a test file.",
}
```

Entre los recursos de Puppet se encuentran, además del fichero, otros que pueden ser familiares, como usuario, grupo, paquete, servicio... Por ejemplo, un recurso paquete puede definirse de una forma similar a:

```
package {'vim':
  ensure  => present,
}
```

Además de la especificación de recursos, Puppet permite la agrupación de varios recursos en una entidad de nivel superior. A esta entidad en la terminología de Puppet se le denomina clase [1] y resulta útil cuando sobre un nodo se quieren aplicar varios recursos, por ejemplo un usuario y el grupo de este ususario:

```
class usergroup {

  user { 'david':
    ensure => present,
    comment => "My user",
    gid => "david",
    shell => "/bin/bash",
    require => Group["david"],
  }

  group { 'david':
```

[1] Aunque el nombre sea el mismo este concepto no es idéntico al usado en la programación orientada a objetos. En Puppet una clase debe verse como una colección de recursos.

```
    ensure => present,
  }

}
```

Asimismo el lenguaje declarativo de Puppet presenta elementos comunes a otros lenguajes de programación, como las variables y las estructuras de control, para mejorar la lógica y la flexibilidad del lenguaje. Estos conceptos pueden verse aplicados en el siguiente manifiesto en el que las variables `$operatingsystem` y `$puppetserver` estarían predefinidas:

```
class sudo {

  package { sudo:
    ensure => present,
  }

  if $operatingsystem == "Ubuntu" {
      package { "sudo-ldap":
      ensure => present,
      require => Package["sudo"],
    }
  }

  file { "/etc/sudoers":
    owner => "root",
    group => "root",
    mode => 0440,
    source => "puppet://$puppetserver/modules/sudo/etc/sudoers",
    require => Package["sudo"],
  }
}
```

Apéndice B
AppScale: Roles y despliegues

A continuación se explica de manera más detallada los distintos roles de la arquitectura AppScale.

En primer lugar tenemos los roles simples:

`Shadow` : Comprueba el estado en el que se encuentran el resto de nodos y se asegura de que están ejecutando los servicios que deben.

`Load balancer` : Servicio web que lleva a los usuarios a las aplicaciones. Posee también una página en la que informa del estado de todas las máquinas desplegadas.

`AppEngine` : Versión modificada de los SDKs de Google App Engine. Además de alojar las aplicaciones web añaden la capacidad de almacenar y recuperar datos de bases de datos que soporten la API de Google Datastore.

`Database` : Ejecuta los servicios necesarios para alojar a la base de datos elegida.

`Login` : La máquina principal que lleva a los usuarios a las aplicaciones App Engine. Difiere del Load Balancer en que esta es la única máquina desde la que se pueden hacer funciones administrativas. Puede haber muchas máquinas que hagan la función de Load Balancer pero sólo habrá una que haga función de Login.

`Memcache` : Proporciona soporte para almacenamiento en caché para las aplicaciones App Engine.

`Zookeeper` : Aloja los metadatos necesarios para hacer transacciones en las bases de datos.

`Open` : No ejecuta nada por defecto, pero está disponible en caso de que sea necesario. Estas máquinas son las utilizadas para ejecutar trabajos MPI.

Además de los roles simples también se proporcionan unos roles agregados que agrupan a varios de los roles simples y que facilitan la descripción de la arquitectura:

`Controller` : Shadow, Load Balancer, Database, Login y Zookeeper.

`Servers` : App Engine, Database y Load Balancer.

`Master` : Shadow, Load Balancer y Zookeeper.

Estos roles pueden usarse en dos tipos de despliegue: por defecto y personalizado. En un despliegue por defecto únicamente se pueden usar los roles `Controller` y `Servers`. En un despliegue personalizado se usan el resto de roles.