Universidad Zaragoza

Proyecto Fin de Carrera
Ingeniería en Informática

Diseño e implementación de un sistema de ejecución de trabajos distribuidos

David Ceresuela Palomera

Director: Javier Celaya

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

Curso 2011/2012
Junio 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, parece que 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 este proyecto se han usado tres ejemplos de infraestructuras distribuidas. La primera de ellas es la 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. Dentro de esta infraestructura distribuida los ejemplos más claros que podemos encontrar son Condor y Torque.

La segunda infraestructura 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 el tercer nivel de la estructura, y por consiguiente, cada vez que uno de los elementos del segundo nivel necesite leer información de la base de datos o modificarla, accederá a este nivel. En esta infraestructura no hay un ejemplo de uso que destaque sobre los demás, pero es tan común que cualquier página web profesional de hoy en día se sustenta en una infraestructura similar a ésta.

La tercera y última es AppScale, una implementación *open source* del App Engine de Google. App Engine permite alojar aplicaciones web en la infraestructura que Google posee. Además de presentar esta funcionalidad AppScale también ofrece las APIs de EC2, MapReduce y Neptune. La API de EC2 añade a las aplicaciones la capacidad de interactuar con máquinas alojadas en Amazon EC2. La API de MapReduce permite escribir aplicaciones que hagan uso del *framework* (o paradigma?) 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. 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 infraestructura necesaria para dar soporte a todas estas APIs ya no es tan sencilla como en los dos casos anteriores. De hecho, las anteriores infraestructuras estarían contenidas en ésta. Hay dos maneras de definir la infraestructura de AppScale. En la primera de ellas se define un despliegue por defecto, en el que un nodo toma el rol de *controller* y el resto de nodos toman el rol de *servers*. El nodo *controller* es el que carga con la responsabilidad de la coordinación y los nodos *servers* son los que llevan cabo la mayor parte del trabajo. La segunda manera de definir la infraestructura es hacerlo a través de un despliegue personalizado. En este despliegue podemos definir con más precisión los roles que queremos que desempeñe un nodo. Entre todos los posibles roles que AppScale ofrece, los más interesantes desde nuestro punto de vista son: *master*, *appengine*, *database*, *login* y *open*.

A lo largo de los últimos años las herramientas de administración de sistemas (o herramientas de gestión de configuración) también han experimentado un considerable avance: con entornos cada vez más heterogéneos y complejos la administración de sistemas complejos de manera manual ya no es una opción. Entre todo el conjunto de herramientas destacan de manera especial Puppet y CFEngine. Puppet es una herramienta de gestión de configuración 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.

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.

1.2 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

La modificación que se ha hecho de la herramienta de configuración Puppet puede considerarse como un concepto novedoso en el campo de las herramientas de configuración. El funcionamiento habitual de este tipo de herramientas consiste en la administración de los recursos de un nodo; el concepto de la administración de un recurso distribuido entre varios nodos no se ha introducido de momento en este tipo de herramientas.

1.4 Tecnología utilizada

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.
- Ruby como lenguaje de programación para la extensión de Puppet.
- 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 [4] para la redacción de esta memoria.

1.5 Organización de la memoria

El resto de este documento queda organizado de la siguiente manera:

Capítulo 2 Estudio de las herramientas e infraestructuras disponibles.

Capítulo 3 Extensión de Puppet para gestión de infraestructuras distribuidas.

Capítulo 4 Diseño de recursos distribuidos específicos.

Capítulo 5 Validación de la solución planteada.

Capítulo 6 Conclusiones.

1.6 Agradecimientos

Agradecimientos

Capítulo 2
Estudio de las herramientas e infraestructuras disponibles

[Revisar]

En este capítulo se aborda el estudio de las distintas herramientas e infraestructuras de interés para la realización del proyecto.

2.1 Elección de la herramienta de virtualización *hardware*

A la hora de hacer una virtualización *hardware* hay varias opciones entre las que elegir. Las más ampliamente usadas son Xen y KVM. La principal diferencia entre ambas es que Xen ofrece paravirtualización y KVM ofrece virtualización nativa. La paravirtualización presenta a las máquinas virtuales una interfaz que es similar, pero no idéntica, al hardware de la máquina física en la que se aloja. Todas las llamadas con privilegios deben ser capturadas y traducidas a llamadas al hipervisor. El sistema operativo de la máquina virtual debe ser modificado para hacer estas capturas.

[Figuras, figuras, figuras]

La virtualización nativa permite hacer una virtualización *hardware* completa de manera eficiente. No requiere de ninguna modificación en el sistema operativo de la máquina virtual, pero necesita de un procesador con soporte para virtualización. KVM está incluido como un módulo del núcleo de Linux desde su versión 2.6.20, así que viene incluido por defecto en el sistema operativo de las máquinas del laboratorio.

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

2.2 Análisis de las infraestructuras de ejecución de trabajos distribuidos

2.3 Elección de la herramienta de gestión de configuración

Dentro de las herramientas de gestión de configuración, se estudió el uso de CFEngine y Puppet. CFEngine es una herramienta de configuración con un lenguaje declarativo en el que se especifican acciones a realizar para las clases. Está programado en el lenguaje C y ofrece un buen rendimiento

y un control detallado de cómo se hacen las cosas. Puppet es una herramienta con un lenguaje declarativo en el que se especifica cuál debe ser el estado de los elementos a configurar. A diferencia de CFEngine, Puppet posee un nivel mayor de abstracción que permite un mejor modelado de los recursos de un sistema. Por ejemplo, Puppet proporciona tipos para modelar usuarios, grupos, archivos, paquetes y servicios. Está programado en el lenguaje Ruby.

A la hora de realizar una extensión de la funcionalidad, se pensó que sería más fácil hacerla en la herramienta que más abstracción proporcionase y en el lenguaje en el que más fácil fuera modelar esta extensión. Por estas razones se eligió Puppet como la herramienta de gestión de configuración sobre la que hacer la extensión.

2.4 Extensión de la herramientas de configuración elegida

Una vez escogida la herramienta sobre la que se haría la extensión, quedaba por determinar cómo realizarla.

La primera opción que se barajó fue la de usar *Faces* de Puppet. *Faces* es una API (*Application Programming Interface*) para crear subcomandos y acciones dentro de Puppet. Analizada a fondo, esta API no proporcionaba una ventaja muy superior a la ejecución de comandos desde la consola del sistema operativo, y no interesaba crear una abstracción que facilitara el trabajo para posteriormente estar usando continuamente la línea de comandos.

La segunda opción que se barajó fue la de la creación de un tipo y un proveedor para ese tipo. Esta opción sí que presenta una ventaja considerable: podemos usar el tipo para modelar la infraestructura distribuida y podemos usar el proveedor para indicar cómo iniciar y mantener esa infraestructura. Esta aproximación se acerca más al modelo que usa Puppet, ya que definimos la infraestructura como si fuera un recurso más de los que posee Puppet. Así pues, esta es la aproximación que se tomó para realizar la extensión.

Capítulo 3
Modelado de configuración automática de infraestructuras distribuidas

[Revisar]

La extensión a la herramienta de configuración Puppet se ha hecho mediante el uso de tipos y proveedores personalizados. Mediante la definición de un nuevo tipo estamos ampliando la cantidad de recursos que Puppet puede administrar; pero para que Puppet sepa cómo administrar ese nuevo recurso debemos proporcionar un proveedor en el que se le indique lo que tiene que hacer.

3.1 Modelado de recursos y configuración en Puppet

Puppet usa un lenguaje declarativo para modelar 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 que se mantenga 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). Por ejemplo, para modelar un recurso de tipo fichero podríamos usar algo similar a:

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

La agrupación de uno o más recursos en un fichero de texto da lugar a un manifiesto. En general, un manifiesto contiene la información necesaria para realizar la configuración de un nodo. Cuando a Puppet se le da la orden de aplicar un manifiesto los pasos que hace son:

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

3.2 Modelado de recursos distribuidos: el recurso *cloud*

El modelado de un recurso distribuido plantea ciertos desafíos al modelo anterior: se puede pensar que para modelar un recurso distribuido basta con que Puppet envíe a cada nodo la configuración necesaria para garantizar el comportamiento deseado, pero, ¿qué pasa cuando ese nodo falla? Si no hacemos nada el recurso dejará de mantenerse en el estado deseado. A la hora de administrar un recurso distribuido hay que asegurarse por lo tanto de que los nodos están operativos y cumpliendo con su función. Asimismo, un recurso distribuido puede presentar elementos comunes para todos los nodos del sistema. Un recurso clásico de Puppet no presenta este problema, ya que o el nodo posee ese recurso, o no lo posee. Se puede hacer una analogía con la programación orientada a objetos: el recurso distribuido sería una metaclase...

Para modelar un recurso *cloud*, se han considerado como indispensables los siguientes parámetros:

- Nombre: Para identificar al *cloud* de manera única.
- Tipo: Para describir qué tipo de *cloud* es. [Modificar]
- Fichero de direcciones IP: Para describir qué dirección IP está asociada a cada nodo del *cloud*.
- Fichero de imágenes de disco: Para asignar a cada nodo la correspondiente imagen de disco duro.
- Fichero de dominio: Para definir una máquina virtual especificando sus características *hardware*.
- Conjunto de máquinas físicas: Para indicar qué máquinas físicas pueden ejecutar las máquinas virtuales definidas.

3.3 Diseño del proveedor

Una vez modelado el recurso *cloud*, queda como tarea proporcionar un proveedor que se encargue de llevar el *cloud* al estado que se le indique desde el manifiesto Puppet. Un *cloud* podrá estar en dos estados: funcionando o parado. Si decidimos que el *cloud* tiene que estar funcionando, el proveedor se encargará de todos los pasos necesarios para llevar al *cloud* a ese estado. A grandes rasgos, estos son los pasos que realiza para ponerlo en marcha:

- Comprobación de la existencia del *cloud*: si existe se realizarán tareas de mantenimiento, si no existe se creará.
- Comprobación del estado del conjunto de máquinas físicas: si una máquina física debe ejecutar muchas máquinas virtuales, existe un riesgo de pérdida de rendimiento.
- Obtención de las direcciones IP de los nodos y los roles que les han sido asignados.
- 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.
- Cuando todas las máquinas virtuales estén funcionando se procede a inicializar el *cloud*.
- Operaciones de puesta en marcha particulares a cada tipo de *cloud*.

De la misma manera, si decidimos que el *cloud* tiene que estar parado, el proveedor se encargará de realizar los pasos necesarios para ello. Estos pasos son:

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

Capítulo 4
Diseño de recursos distribuidos específicos

4.1 Uso de la extensión aplicado a la infraestructura AppScale

4.1.1 Manifiesto de recurso distribuido

4.1.2 Fichero de roles

4.2 Uso de la extensión aplicado a la infraestructura Torque

4.2.1 Manifiesto de recurso distribuido

4.2.2 Fichero de roles

4.3 Uso de la extensión aplicado a la infraestructura web de tres niveles

[Revisar]

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 arquitectura puede complicarse añadiendo más niveles y añadiendo más elementos a los niveles existentes.

Para demostrar la validez del modelo desarrollado, además de AppScale también se puede controlar una infraestructura de servicios web. En el ejemplo se ha validado una arquitectura que consta de un balanceador de carga, dos servidores web y un servidor de bases de datos. Como balanceador de carga se ha usado nginx y como servidor de base de datos se ha usado MySQL. La creación de la página web se ha hecho usando el framework Sinatra sobre el servidor web WEBrick. Todos ellos corren sobre máquinas con sistema operativo Ubuntu.

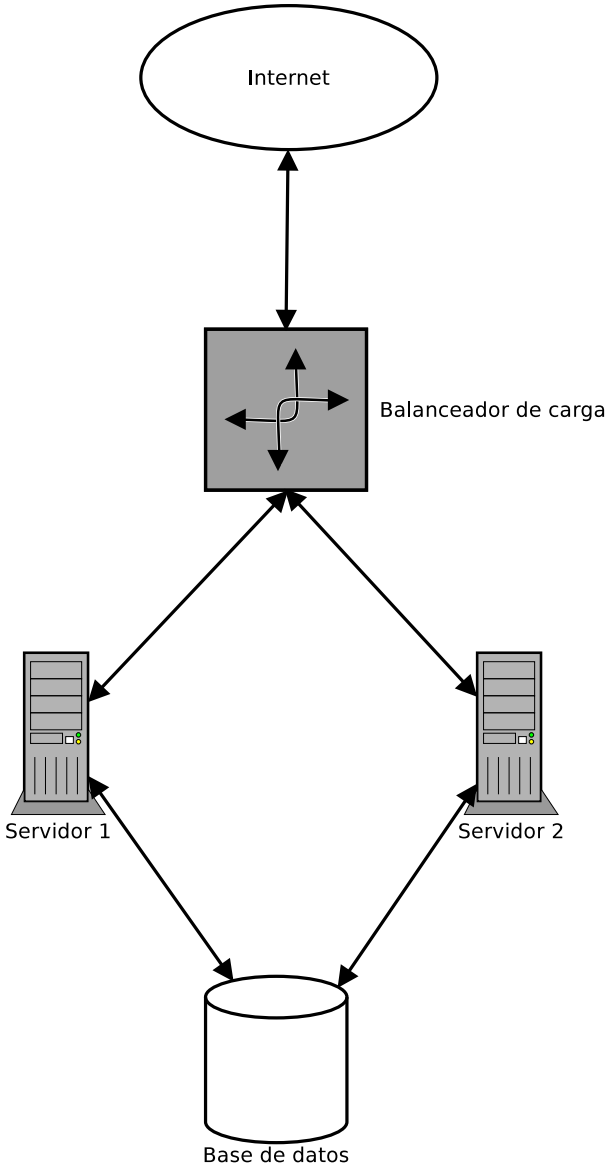

Figura 4.1: Arquitectura web de tres niveles.

4.3.1 Manifiesto de recurso distribuido

La sintaxis en el metamanifiesto es fundamentalmente similar a la utilizada en el ejemplo de AppScale: el único campo que cambia sustancialmente es el type, que pasa de tener valor `appscale` a tener valor `web`. El resto de campos se comportan como lo hacían en el ejemplo de AppScale: el campo file contiene el fichero de descripción de roles, el campo images contiene los discos duros de las máquinas virtuales, el campo domain contiene el fichero de descripción de la máquina virtual, el campo pool contiene el conjunto de máquinas físicas a usar y el campo ensure contiene el estado en el que queremos que quede el cloud.

```
cloud {'mycloud':
   type    => "web",
   file    => "/etc/puppet/modules/cloud/files/web-simple.yaml",
   images  => ["/var/tmp/dceresuela/lucid-web.img", "/var/tmp/dceresuela
      /lucid-db.img"],
   domain  => "/etc/puppet/modules/cloud/files/mycloud-template.xml",
   pool    => ["155.210.155.70"],
   ensure  => running,
}
```

4.3.2 Fichero de roles

El contenido del fichero de especificación de roles sí que poseerá valores distintos a los que tenía el ejemplo de AppScale ya que estamos describiendo una arquitectura distribuida distinta. Los nuevos roles que pueden desempeñar las máquinas son:

`balancer` La máquina que desempeñará el rol de balanceador de carga.

`server` La lista de máquinas que desempeñarán el rol de servidor web.

`database` La máquina que desempeñará el rol de servidor de base de datos.

Un ejemplo completo del fichero de especificación de roles tendría un contenido similar a éste:

```
---
:balancer: 155.210.155.175
:server:
- 155.210.155.73
- 155.210.155.178
:database: 155.210.155.177
```

Capítulo 5
Validación de la solución planteada

x Máquinas físicas
y Máquinas virtuales
Servidor DNS
AppScale: 1 maestro, dos esclavos
Torque: 1 maestro, dos esclavos
Web: 1 balanceador de carga, 2 servidores web, 1 base de datos

Capítulo 6
Conclusiones

Conclusiones.

Bibliografía

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

[2] 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.

[3] 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.

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

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

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

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