Automating Provisioning of Complete Software Stack in a Grid Environment

Bikash Agarwalla*
Georgia Institute of Technology
801 Atlantic Drive NW
Atlanta, GA 30332 USA
bikash@cc.gatech.edu

Vanish Talwar, Sujoy Basu, Raj Kumar
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304 USA
{vanish.talwar,sujoy.basu,raj.kumar}@hp.com

Abstract

With scaling of data centers, clusters, and grids, it is going to be increasingly difficult to download, configure, install, update, and manage the software stack on the constituent nodes. Automation of this process is needed for the system to scale beyond a few hundred nodes. We present system architecture for a middleware service that performs this task in an automated manner. We propose that our architecture can be implemented as a grid service as part of the globus toolkit. Furthermore, we demonstrate the usefulness of the architecture through a prototype implementation which automates the whole process by integrating various existing solutions along with building intelligence into the middleware to handle such tasks.

1 Introduction

Grid Computing has its roots in solving compute intensive problems. Traditional use of grid has been in solving intensive medical research problems, weather forecasting and scientific visualization applications. But recently, researchers have widely acknowledged the use of grid for interactive applications and data centers as well. In a data center environment, grid computing can be used to provide various features such as security, virtualization, naming, and discovery.

With scaling of data centers, clusters, and grids, we propose that it is going to be increasingly difficult to install, configure, update, and manage the software stack installed on the machines. For a small data center, most of the configuration management tasks can be performed manually, but when the size of the data center increases to a few hundreds or thousands of nodes, manually performing the configuration tasks become practically impossible. More complication arises due to the heterogeneity in the software and hardware configuration. If all the machines in the grid have same hardware and software configuration, a simple script may be enough in configuring and maintaining them. Differences in the hardware configuration may come due to newly bought machines or upgraded machines in the data center. The software configuration may become different because of different customer needs. All of the different configurations are necessary if a data center is to be used efficiently with minimal financial impact. Because of these heterogeneity in the configurations, a higher level representation of configuration attributes and validation is needed. Finally, the system should also have monitoring and fault recovery services and security features to protect the resources from malicious use.

Thus, we argue that automation of the whole process of installation and deployment of complete software stack is needed for the system to scale beyond a few hundred machines. Many design choices exist, however, we describe a general system architecture that can take care of the issues mentioned above. In our prototype implementation, we focus on automating the whole process by integrating various existing solutions along with building intelligence into the middleware to handle such tasks. We have identified following set of tasks as crucial in automating provisioning of complete software stack in a grid environment.

- Software Install: Software installation step includes installation of OS as well as the application packages. We consider a data center providing resources for hosting e-commerce applications. In this case, the various tiers of an n-tier

*Work done during internship at HP Labs. Current affi liation: College of Computing, Georgia Tech, Atlanta, GA.

e-commerce application should be installed in addition to the OS. The system architecture should take care of dependencies between various tiers. For the system to be scalable, it should also automate this process with support for remote administrations.

- Version Upgrade: This process involves applying routine upgrades to already installed applications on various machines. The upgrades should be done seamlessly with minimal system downtime.
- Software Configuration: The software installed on the machines may need to be configured properly before they can be deployed. A web server typically is configured to run on a specific port and access data from specific directory. Similar configuration steps are needed for application servers and database servers. Some of the configuration aspects may be related to security while others may be related to runtime parameters. Our system architecture performs these software configurations automatically based on higher level policies specified typically by a grid administrator.
- Deployment: This step involves running the applications on the specific nodes of the data center. The system should take care of dependencies between various applications and deploy them in order.
- Life-cycle Management: The life-cycle management component typically performs the monitoring of the deployed applications and take corrective actions based on the system load, or application failures. The various decisions related to life-cycle management are taken based on policies. Life-cycle management task is typically performed in conjunction with the resource allocator.
- Termination: The deployed applications may need to be terminated based on the user request or by the system as part of life-cycle management.
- Software Uninstall: Software uninstall may be needed driven by the user request. It may also be necessary to uninstall the software installed on a machine for efficient utilization of resources in the grid.

Separate tools exist which address some of the issues mentioned above, but none of the existing solution is capable of automating the whole process in an intelligent way. We have taken into account all of the above issues in coming up with a solution for automatic provisioning of complete software stack (operating system as well as an e-commerce application) in a grid environment. Our contributions in this paper are as follows: 1) we present a system architecture that can be used to automatically install and deploy the complete software stack in a grid environment, 2) we propose that our architecture can be implemented as a grid service as part of the globus toolkit, and 3) we demonstrate the usefulness of the architecture through a prototype implementation.

The rest of the paper is organized as follows: In section 2 we describe our system architecture. In section 3 we discuss our system implementation in detail and present related works in section 4. Finally, we present our conclusions in section 5.

2 Architecture

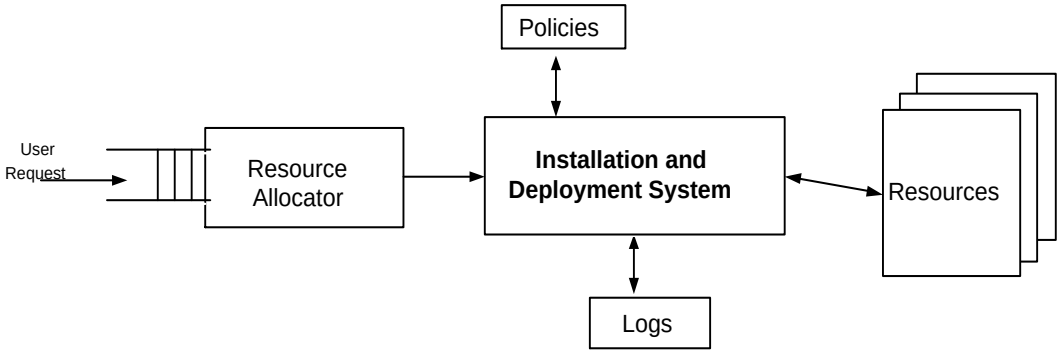

Figure 1. System Architecture

Figure 1 demonstrates our system architecture. The user submits request for deploying an e-commerce application through the grid portal. The resource allocator takes the requests from the input queue and assigns resources based on scheduling algorithms. Details of our resource allocator system can be found from [11].

In this paper, we focus on the installation and deployment system. Based on the resource assignments, the installation system performs OS installation on the target nodes. The system takes decisions based on policies specified by the grid administrator. Based on policy, the system will decide the packages that should be installed on the target nodes.Once the installation is complete, th e-commerce application will be deployed using the deployment system.

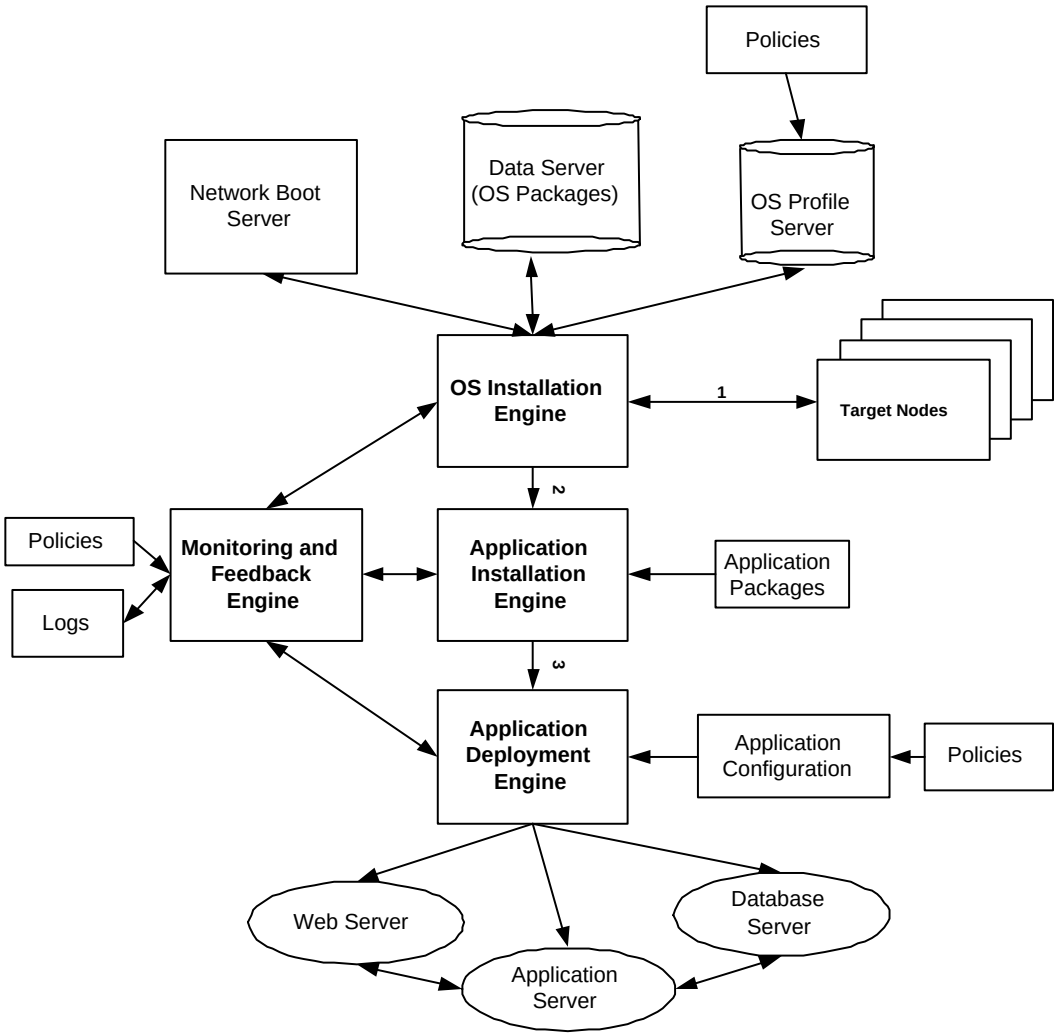

Figure 2. Installation and Deployment System Architecture

Figure 2 illustrates the architecture of the installation and deployment system in detail. There are four main components of the system, OS installation engine, application installation engine, application deployment engine and the monitoring engine. Each of these components can be implemented as a grid service.We describe these various components and their interfaces in detail.

The OS installation engine performs the automated OS installation. It contacts the network boot server for bootstrapping the installation process. The network boot server will typically be a bootp server used for performing network installation. The data servers will contain the installation packages and the kernel images for initiating the installation. The data servers are distributed for load balancing purpose. The OS profiles will determine the particular softwares that will be installed on a machine. These profiles are generated automatically based on policies. For example, all machines in the product group of an enterprise may require a specific machine configuration which is different from the machines in the research and development

group. These policies can be used to customize the grid environment. In order to integrate with the application installation engine, the OS installation engine performs the following steps.

- OS installation engine sends a completion message to the application installation engine so that application installation can be initiated.
- OS installation engine installs a specific application at the end of OS installation. This specific application is responsible for installing and deploying the e-commerce application. In our implementation, we used SmartFrog[7] as the specific application.
- OS installation engine should send progress information to the monitoring engine periodically so that any failure can be identified and corrective action can be taken.

Implementing OS installation engine as grid service will require it to exchange the above set of messages in order for it to be correctly integrated with the other components of the installation and deployment system.

Once OS installation engine has completed its task, the application installation engine takes over the target node and is responsible for installing the applications. In this paper we focus on e-commerce application installation in a grid environment. The application installation engine will consult the appropriate application packages (webserver, app server and database server) and install it on the allocated machines. The application installation engine integrates with the application deployment engine through the exchange of following messages:

- Application installation engine sends a completion message to the application deployment engine so that the application deployment can be initiated.
- Application installation engine should send progress information to the monitoring engine periodically so that any failure can be identified and corrective action can be taken. Failure in this case can be related to application packages being unavailable or improper configuration of the target node. The monitoring engine can instruct the application installation engine to install all the dependent packages before proceeding with the application installation. The monitoring engine can also decide to chose another target node for the application installation in case of node failure. The monitoring engine will contact the resource allocation system for this service.

Once the application is installed, the application deployment engine is responsible for starting and stopping the application. The deployment engine is also responsible for managing the life-cycle of the running applications. This requires continuous monitoring of the running applications to identify system load and the number of instances of the applications. System wide policy will dictate the actions that need to be taken incase the system load exceeds a threshold. Another example of policy that a typical e-commerce application may specify is to have certain instances of the various servers running at any time depending on the load. Such policies can be enforced by the application deployment engine as part of life-cycle management. The various configuration parameters related to application deployment and life-cycle management are read from the application configuration module. These application configurations are generated at run time based on policies. In our implementation, the application deployment system is used to deploy the webserver, application server and the database server. The application deployment system is integrated with the other components through the exchange of following messages:

- Application deployment engine should send progress information periodically to the monitoring engine.
- The life-cycle management component of the application deployment engine should notify the monitoring engine in case of any failure. Failure can be due to changes in the configuration of the target node which prohibit deployment of the application. The monitoring engine can try to correct the node configuration or may decide to deploy the application on a different target nodes depending on policy. Deploying the application on a different node will require notifying all the dependent applications so that connectivity is maintained between the different components of the e-commerce application.

The monitoring engine is responsible for monitoring the progress of the different system components. It monitors the progress of the OS-installation engine, the application installation engine and the application deployment engine. The system progress is written to the log files. In case of failures of any particular component, the monitoring engine provides feedback based on various policies as mentioned above.

The above system architecture can be used to deploy any OS (linux, windows, solaris) and any e-commerce application in an automated manner. We have identified the interfaces between the various engines so that the OS installation engine, application installation engine and the application deployment engine can be implemented as grid services. section Implementation In this section we describe our system implementation for automating OS as well as e-commerce application installation and deployment. We chose to install redhat linux OS and a 3-tier application in an automated manner. The 3-tier application consists of a web server, an application server, and a database server. We have used kickstart [10] with preboot execution environment (PXE) [6] as the underlying technology for automating OS installation in a grid environment. The 3-tier application is installed and deployed using SmartFrog [7]. Though we demonstrate the usefulness of the system through these specific examples, our architecture is general enough so that it can be used to perform similar tasks for other applications and OS as well. The details of each of the system components are described below.

2.1 OS Installation

In a large data center environment, it is not possible to do OS installation using floppy drive or CD-ROM as is traditionally done. Even when OS is installed through the network, the bootstrap method traditionally used makes use of a floppy with the minimal linux kernel.This set up is also not scalable as the number of machines increases beyond a few hundreds. Our system uses a combination of PXE and Kickstart to completely automate the process of OS installation without the need for any floppy or cd-rom.

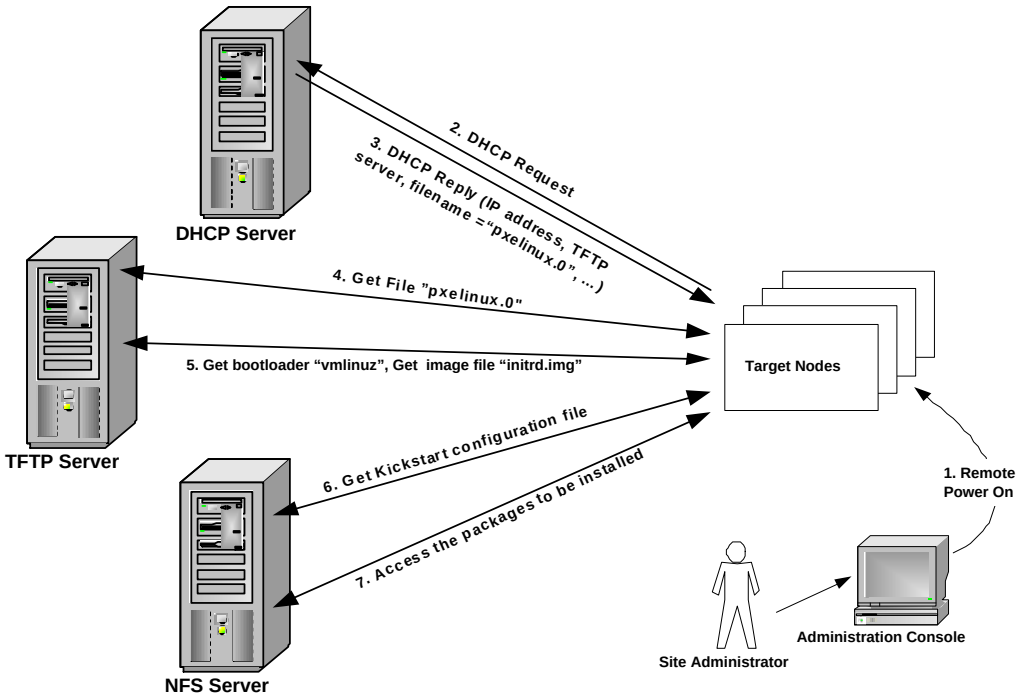

Figure 3. System Architecture for OS installation

The figure 3 describes our system architecture for automating the operating system installation. The system utilizes a DHCP Server, TFTP Server and NFS server. We assume that the target nodes do not have any software installed on it, for example, these may be the newly bought machines in the data center. The target nodes use the various servers mentioned above for getting the different pieces required for OS installation. The servers may all be located on one machine or they may be distributed. There may be multiple servers for load sharing in our system.

The DHCP server is used to assign an ip address to the target nodes. the server receives the request for ip address from the target node during the network boot, and responds with the ip address, the name of the TFTP server for downloading the initial boot image, and the name of the network boot program.

The TFTP server is used to provide various files to the target nodes. The target node first gets the network boot program(NBP) *pxelinux.0* from the TFTP server and starts booting from it. The NBP gets the linux kernel and all the image file

from the TFTP server and starts loading the kernel. The kernel is passed, as argument,the location of the kickstart file on the network (NFS server).

The NFS server is used to provide the kickstart file to as well as the RPM packages that need to be installed to the target nodes. The target node gets the kickstart file from the NFS server and starts using it for automated OS installation. The kernel uses the kickstart file to find the location of the packages on the NFS server.

Once the target node is powered up by the system administrator through an administration console, all the installation steps are carried out without any human input.

The figure 4 describes the sequence of steps that are carried out during the OS installation step on a single target node. When the target node is rebooted to perform network install, a DHCP request is sent to the DHCP server. The DHCP server identifies the new machine based on its MAC address and responds with an IP address. The DHCP server also sends the location of the TFTP server, and the location of the network boot program on the TFTP server. The client then contacts the TFTP server and fetches the network boot program (PXE bootloader) and starts executing it. The PXE boot loader gets the configuration file from a specific location on the TFTP server. Based on the information in the configuration file, the PXE boot loader fetches the appropriate linux kernel from the TFTP server and starts loading it. The kernel is also passed various arguments based on the configuration file. The linux kernel gets the initial RAM disk image from the TFTP server, accesses the kickstart file from the NFS server and starts the kickstart installation. The kickstart installation fetches the various RPM packages from the NFS server. No human input is required during this whole installation process.Once the installation completes, the system reboots and comes up with the linux console.

This section briefly describes the details of the various steps that are performed in order to carry out the automated OS installation. The following steps were carried out:

- BIOS configuration at the target node
- Gateway machine configuration
- DHCP server configuration
- Network Boot Program
- TFTP server configuration
- NFS server configuration

We will describe each of these steps in detail below.

2.1.1 BIOS configuration

The new target node needs to be configured for booting over the network before the installation can proceed. This step may require changing the BIOS configuration to enable booting over the network. In our set up, we enabled the network booting from the boot option section of the BIOS configuration. After this step has been performed, the target node only needs to be powered on once in order for the installation to proceed. Our system can also use remote power up facilities if available.

2.1.2 Gateway machine configuration

In our set up, we used a gateway machine with two network cards, one for connecting to the local data center machines and the other for connecting to the internet. The external interface is very useful for remote administration, monitoring and maintainance. The internal network card was configured to have an IP address internal to the network. This configuration was done by using the following commands:

```
bash# /sbin/ifconfig eth1 192.168.0.1 netmask 255.255.255.0
```

2.1.3 DHCP configuration

We downloaded and installed the DHCP server. The details of the dhcp server installation steps are listed below.

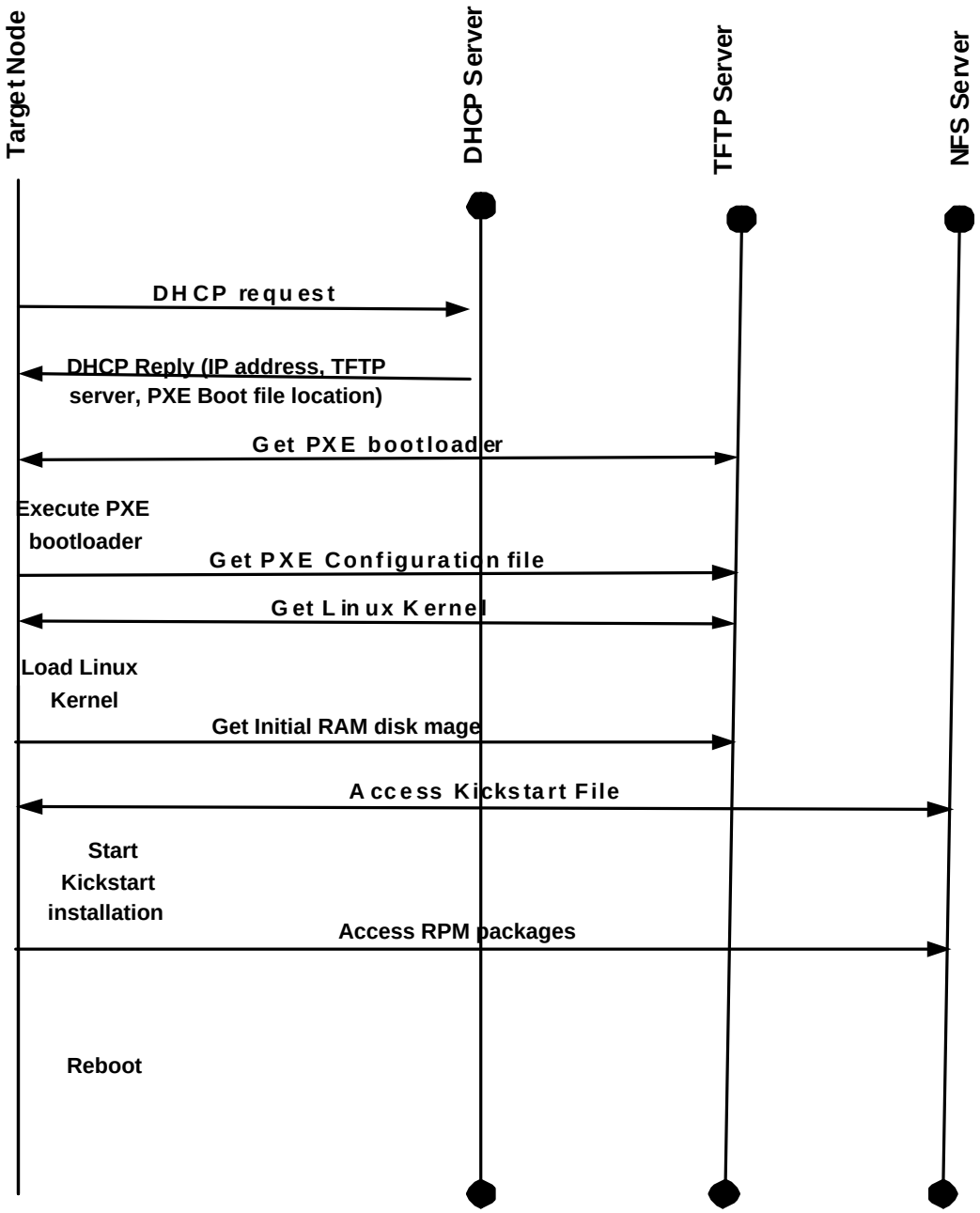

Figure 4. Sequence Diagram

```
1. Get DHCP source from ftp://ftp.isc.org/isc/dhcp/dhcp-3.0pl2.tar.gz
2. untar it.
3. bash# ./configure; make
4. bash# su
5. bash# make install
6. Create an empty leases file: bash# touch /etc/dhcpd.leases
7. Create /etc/dhcpd.conf according to the template.
8. Start the dhcp server: bash# /usr/sbin/dhcpd
```

```
# /etc/dhcpd.conf
deny unknown-clients;
not authoritative;

option domain-name              "hpl.hp.com";
option domain-name-servers      192.168.0.1;
option subnet-mask              255.255.255.0;

allow bootp;
allow booting;

option ip-forwarding    false;  # No IP forwarding
option mask-supplier    false;  # Don't respond to ICMP Mask req

subnet   192.168.0.0 netmask 255.255.255.0 {
  option routers        192.168.0.1;
}

subnet 15.9.0.0  netmask 255.255.0.0 {
#  option routers     15.9.72.237;
}


group {
  next-server     192.168.0.1;          # name of your TFTP server
  filename    "/tftpboot/pxelinux.0";       # name of the bootloader program

  host quark6 {
       hardware    ethernet    00:10:83:02:97:BE     ;
       fixed-address      192.168.0.11    ;
  }
}

ddns-update-style ad-hoc;
```

Figure 5. DHCP Configuration

The figure 5 illustrates a sample configuration file for DHCP server. This DHCP server identifies two subnets corresponding to the two network cards installed on the machine on which it is running. The external network card is connected to the internet and belongs to subnet *15.9.0.0*. The internal network card is connects to target nodes on which OS installation needs to be performed. This DHCP server only responds to requests from the internal network. The **next-server** attribute specifies the location of the TFTP server which in our set up lies on the same machine as the DHCP server. The **filename** attribute

specifies the location of the network boot program on the tftp server. The DHCP server in our case, assigns fix IP address based on the MAC address of the machine. The IP address can also be assigned automatically from a range of addressses by configuring the DHCP server. In our particular implementation, the DHCP server needs to know the MAC address of each of the target node. Our system architecture also supports automatic detection MAC addresses by writing appropriate scripts. The DHCP server may also point different clients to different TFTP server for load sharing.

2.1.4 Network Boot Program

We used PXE as our network boot program. It can be obtained from the syslinux package [**?**]. We compiled the syslinux package and copied the NBP *pxelinux.0* in the */tftpboot* directory on the TFTP server. When the network boot program is loaded, it gets the configuration file from the TFTP server. The PXE boot loader looks for several configuration file in a specific directory on the TFTP server in a pre-determined order and picks the first file it finds. The filename it looks for are determined by the IP address of the target node it is running on. Based on the configuration file information, it gets the linux kernel and starts executing it with appropriate arguments. Figure 6 illustrates a sample configuration file.

```
#PXELINUX CONFIG FILE
# boot from the network
default linux
serial 0,38400n8
label linux
  kernel    vmlinuz-7.2
  append ramdisk_size=32000 initrd=          initrd-7.2.img     \
            network      ks= nfs:192.168.0.1:/tftpboot/data/kickstart/ks-7.2.cfg
```

Figure 6. PXE Configuration File for Network Boot

2.1.5 TFTP Configuration

We used *tftp-hpa* which was started from xinetd. The TFTP server is populated with the various files that are needed for installation. In our set up, the TFTP server is used to serve the bootloader program *pxelinux.0*, the linux kernel *vmlinuz-7.3*, the initial RAM disk *initrd.img-7.2*, and the pxe configuration file. The TFTP server maintains a configuration file for each client. It also maintains a default configuration file which lets the client machine boot from the hard disk. Once the OS installation is complete, the system automatically updates the configuration file for the client to point to the default configuration so that when the machine is restarted, it will load the newly installed OS.

The section below lists the sequence of steps that were carried out in order to configure and install the TFTP server.

```
1. Get TFTP package from http://www.kernel.org/pub/software/network/tftp/
2. Compile and install the package
3. Configure the TFTP server as xinetd process by creating a file
   /etc/xinet.d/tftp with the following lines:
   #/etc/xinet.d/tftp
   service tftp
   {
     socket_type = dgram
     protocol = udp
     wait = yes
     user = root
     server = /usr/sbin/in.tftpd
     server_args = -v -r blksize -s /tftpboot
     disable = no
     per_source =  11
     cps = 100 2
   }
```

```
4. bash# /etc/init.d/xinetd restart
5. Populate /tftpboot with the following files
   - The bootloader program pxelinux.0
   - A compressed linux kernel vmlinuz-7.3
   - An initial RAM disk image initrd.img-7.3
   - a subdirectory pxelinux.cfg/ with two actual files
       - default.netboot-7.3 and
       - default
   - one symbolic link for each target node poiniting to the
     default.netboot-7.3 file
```

2.1.6 NFS Server Configuration

The NFS server is used to host the RPM packages and the kickstart configuration file for the various clients. The kickstart configuration file contains the choices that are to be made during the OS installation. It also contains the location of the packages that are to be installed.

```
# Kickstart File
lang en_US
langsupport --default en_US
keyboard us
mouse generic3ps/2
timezone --utc EST5EDT
rootpw --iscrypted $1$L3ö0xDbà$ycEAX7PADvOC/4jlO9PAZ1
reboot
#clearpart --all --initlabel
bootloader
install
nfs -- server 192.168.0.1 -- dir /redhat7.2
#Clear only linux partitions from the disk
clearpart --linux
part /boot --fstype ext3 --size=50 --ondisk=sda
part / --fstype ext3 --size=10 --grow --ondisk=sda
part swap --size=1000 --grow --maxsize=2000 --ondisk=sda
network --bootproto dhcp
auth --enablemd5 --useshadow
firewall --disabled
skipx
%packages

%post
```

Figure 7. Kickstart Configuration

A sample kickstart file is shown in figure 7. This configuration file informs the location of the RPM packages on the NFS server. It also gives information about disk partitions, firewall, the various packages to be installed in addition to other choices that are made during installation.

In our particular implementation all the above servers were running on a single machine with two network card, one for connecting to the internal network and another for connecting to the internet. The target nodes were on the internal network. The installation progress can be visualized through this external network interface on the server machine. As part of the post

installation process, we performed the installation and configuration of smartfrog [7]. We use SmartFrog in our system to perform application installation and deployment.

2.2 Application Installation and Deployment

This section describes our system implementation for performing a 3-tier application installation and deployment. The automation process goes through the sequence of downloads, installation, configuration, deployment, life cycle management and termination for the 3-tier application. In our set up, we chose to follow these steps for the application server (JBoss) [8], the database server (postgres) [2], and a servlet application (guestbook) built by us. The *guestbook* application allows a user to enter his name and some messages. The messages are stored in the database and the system responds with all the messages that are stored as well as the number of times the user has logged in after querying the database. This servlet application use *JDBC* [1] api's to connect to a database. In our particular testbed, we chose not to install and configure the webserver component, though the system can easily support it as well.

We use SmartFrog [7] as the underlying technology for automating the installation and deployment of application. SmartFrog is a framework for describing, deploying, igniting and managing distributed applications. SmartFrog focusses on the higher level of the configuration task like application service configuration and ignition. The SmartFrog framework consists of a configuration description language, component model, and the deployment infrastructure. Its life cycle management infrastructure is able to deal with a software system as a single entity even though it may be running over multiple, distributed nodes.

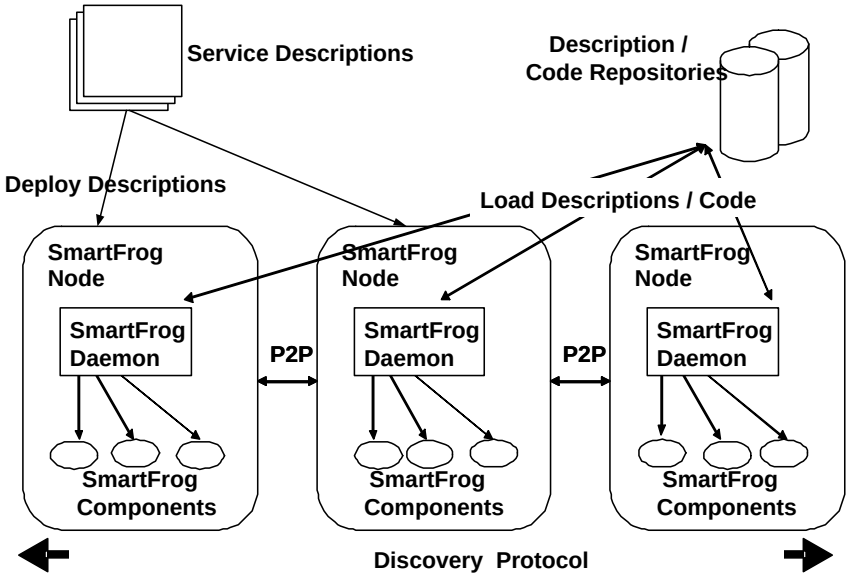

Figure 8. SmartFrog Architecture

The figure 8 describes the smartfrog architecture. The various components are described using smart frog description language. The description includes details like which particular component to deploy on what particular nodes and what are the configuration parameters needed by the component. The smartfrog descriptions can be deployed anywhere into the deployment system. The descriptions and component code can be loaded locally or remotely. The smartfrog daemon running on each node is responsible for deploying and terminating the various components. The daemon loads the configuration description and the component implementation code from the repositories and deploys the component at appropriate nodes. The smartfrog daemon running on various nodes discover and communicate using peer to peer protocols.

We find smartfrog to be a very general and useful framework which is capable of maintaining the configuration of large

scale, diverse and dynamic environment. Appropriately built components will reduce the time needed to automate the configuration tasks considerably. We have utilized smartfrog's built in components whereever possible and have written additional components where necessary.

In order to ensure that the newly installed target node can run smartfrog components, we install smartfrog package as part of the os installation and configure the target node so that the smart frog daemon starts up at boot time. All of this is done through kickstart configuration. As a result, when the OS installation is complete, the target node comes up with smart frog installed, and the smartfrog daemon running and configured to deploy applications.

2.2.1 Application Installation

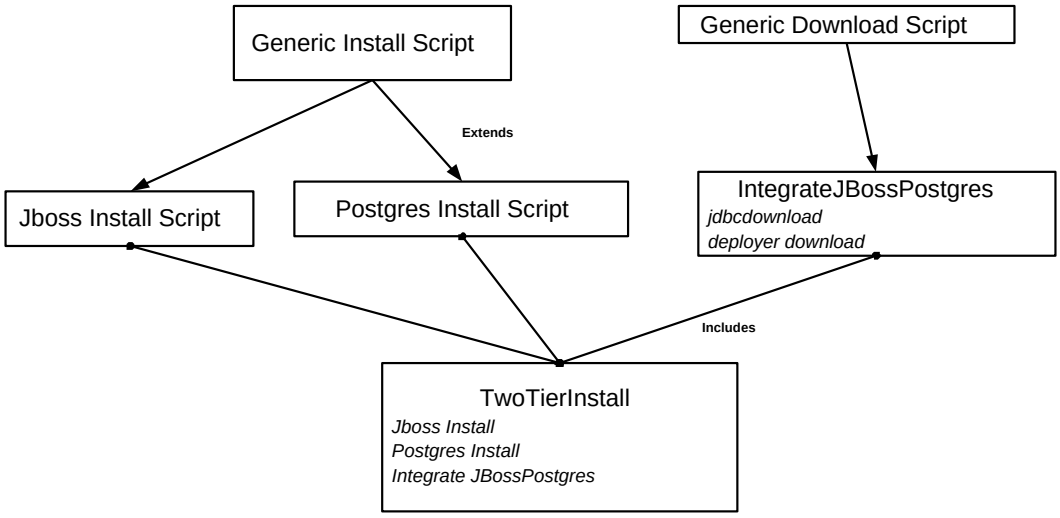

Figure 9. Smart Frog components for installation and integration

The figure 9 illustrated the various components that were written in order to automate the installation and integration process of jboss and postgres.

Download In our set up, the application packages are downloaded from a web server. We extended the smartfrog in-built component *GenericInstallScript* to download jboss and postgres applications. The code segment in figure 10 describes the component description used to download postgres and jboss.

```
#include "org/smartfrog/examples/dynamicWebServer/sf/
genericInstallScript.sf";
 postgresInstall extends genericInstallScript {
   webServerHost "quark4.hpl.hp.com";
   webServerPort 80;
   tarLocation "/~bikash/postgresql.tar.gz";
   installLocation "/home/bikash/Automation/Install/";
   file "postgresql.tar.gz";
   name "postgresql";
   untarFile extends concat {
       -- "tar -zxvf";
       -- ATTRIB file;
   }
   // instuctions for doing installation.
   ...
 }
```

SmartFrog Description for Postgres Download

```
#include "org/smartfrog/examples/dynamicWebServer/sf/
genericInstallScript.sf";
 JBossInstall extends GenericInstallScript {
   webServerHost "quark4.hpl.hp.com";
   webServerPort 80;
   tarLocation "/~bikash/jboss.tar.gz";
   installLocation "/home/bikash/Automation/Install/";
   file "jboss.tar.gz";
   name "jboss";
   untarFile extends concat {
       -- "tar -zxvf";
       -- ATTRIB file;
   }
   // instuctions for doing installation.
   ...
 }
```

SmartFrog Description for Jboss Download

Figure 10. Smart Frog components descriptions for downloading applications

The *postgresInstall* component above extends the existing component *genericInstall* by redifining some attributes with specific values suited for downloading. It redefines the *webServerHost*, *webServerPort* and *tarLocation* attributes to the values needed for downloading the tar-compressed file. It also defines the installation directory location through *installLocation* and the destination file name through *file* attributes. The *name* attribute is used to identify the component for debugging purposes. The *untarFile* attribute uses smart frog predefined component *concat* to concatenate the pieces of the commands. The *ATTRIB* key word is used to have the actual value of *file* placed in its place in the type resolution phase of smart frog.

concat is a predefined function that comes with the smart frog library. All the other attributes that are defined above are interpreted by the *genericInstall* component.

The component description for downloading jboss is quite similar to Postgres as is illustrated in the figure 10.

Installation In order to perform the installation steps for jboss and postgres, we redefined the ”installScript” component which is interpreted by the ”genericInstallScript” component mentioned in the last section.

```
postgresInstall extends genericInstallScript {
  ...
  installScript extends append {
    -- extends vector {
         -- ATTRIB cdHome;
         -- ATTRIB configureScript;
         -- ATTRIB makeScript;
         -- ATTRIB makeInstall;
      }
      ...
  }
  cdHome extends concat {
    -- "cd ";
    -- ATTRIB home;
  }
  configureScript "./configure --prefix=/home/bikash/Automation/Install
                                      /postgresql-7.3.4/";
  makeScript "gmake";
  makeInstall "gmake install";
  ...
}
```

The *genericInstallScript* interprets the *installScript* component as a vector and executes each string entry in the vector as a command from a bash shell invoked from within the java component. By including appropriate commands, the installation steps of postgres are performed as shown above. In the above template, *append* is an inbuilt component in smart frog that takes a set of vectors and append all the vectors elements to create a new *vector*. The *cdHome* is the command to change the working directory to the installation directory. *configureScript* is the command to perform preinstall configuration for postgres. The *makeScript* and *makeInstall* are the commands used to compile and install postgres respectively.

The jboss installation is complete once all the files from the tar-compressed archive are extracted. So, the installScript for jboss does nothing.

Post-install configuration Some application specific configuration may be needed once it is installed In the case of postgres database, a directory for holding the data need to be created and initialized before anyone can use the database. Also, note that setting up accounts for creating and accessing database, creation of database itself may need to be performed too. We do not show the automation of the above step here though it can also be done from the smartfrog components.

```
postgresInstall extends genericInstallScript {
  ...
  installScript extends append {
    ...
    -- ATTRIB postInstallconfigure;
```

```
  }
  ...
  postInstallconfigure extends vector {
   -- "mkdir data";
   -- "bin/initdb -D data";
  }
  ...
}
```

The code template above shows the initialization steps for postgres. The commands inside *postInstallconfigure* specify the steps that need to be performed. In the case of JBoss, no post install configuration was needed.

Integration of jboss and postgres In our implementation, we also performed certain configuration steps to integrate jboss with postgres. We assumed that a webserver is serving the required files which include the postgres jdbc driver and a xml descriptor for deploying it in jboss. We wrote smartfrog components that download the required files and puts it into proper location. The download of various files can be performed in parallel. Smartfrog deployment model allows parallel execution of these various steps as well.

Another smartfrog component was written which was responsible for combining the above components together in order to perform the installation of the three tier application. It follows the sequence of installing jboss, installing postgres and integrating the two.

2.2.2 Application Deployment

The figure 11 illustrates our smartfrog component architecture to perform the deployment. The **GeneriApp** component

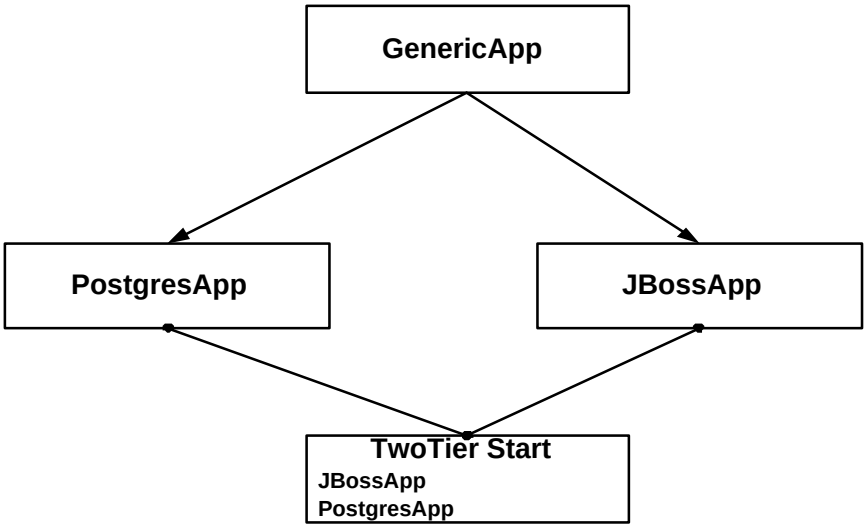

Figure 11. Smart Frog components descriptions for deployment and lifecycle management

description takes care of starting, stopping and life cycle management of any generic application. By appropriately defining the specific attributes, the specific instances such as *PostgresApp* and *JBossApp* can deploy the specific applications postgres and jboss respectively. The following template describes the **GenericApp** component.

```
genericApp extends Compound {
  sfClass "genericImpl";
  location extends concat {
    -- "/home/bikash/JBoss/jboss/";
    filepath "";
```

```
    }
    appName "JBoss";
    runScriptLocation extends location {
      filepath "bin/run.sh -c all";
    }
    stopScriptLocation extends location {
      filepath "bin/shutdown.sh";
    }
    executableName "run.sh";
    ...
}
```

The *sfClass* attribute specifies the class name that is used to deploy this component. This class is responsible for interpreting the attributes that are defined in this component and taking appropriate action. The attribute *appName* defines the name of the application and is used for printing debugging messages. The *runScriptLocation* attribute defines the command used to start the application. The *stopScriptLocation* specifies the command that needs to be executed for stopping the application. The attribute *executableName* specifies the process name used to identify the processes started by the component. It is useful for maintaining the life cycle of the started application and making sure that at least one instance of the application server and the database server is running.

The **JBossApp** and **PostgresApp** components extend the **GenericApp** component. The *PostgresApp* component description is included below:

```
#include "genericApp.sf"
postgresApp extends genericApp {
  home "/home/bikash/Automation/Install/postgresql-7.3.4/";
  sfProcessHost "quark4.hpl.hp.com";
  location extends concat {
    -- ATTRIB home;
    filepath "";
  }
  appName "PostGres";

  dataSource extends location {
    filepath "data/";
  }
  runScriptLocation extends location {
    filepath extends concat{
      -- "bin/pg_ctl -o -i start -D ";
      -- ATTRIB dataSource;
    }
  }
  stopScriptLocation extends location {
    filepath extends concat{
      -- "bin/pg_ctl stop -D ";
      -- ATTRIB dataSource;
    }
  }
  executableName "postmaster";
  name "PostMaster";
}
```

In tha above example, the attributes *home, location, dataSource* are used as temporary place holders since they are not interpreted by the component *GenericApp*. The *sfProcessHost* attribute defines the name of the machine on which postgres is going to be deployed. The *appName, runScriptLocation, stopScriptLocation, executableName and name* attributes are properly defined as required by postgres.

Tha same template *GenericApp* template is used for deploying jboss aswell. The smartfrog description file *JBossApp* is very similar to *PostgresApp*. The smart frog description *twotier* describes how the complete twotier application is going to be deployed.

Another component description was written which will deploy the servet application *guestbook*.

```
#include "GenericDownloadScript.sf";
servletApp extends Compound {
  WebHost "quark4.hpl.hp.com";
  WebPort 80;
  JBossHome "/home/bikash/Automation/Install/jboss-3.2.2/";
  name "guestbook";
  -- extends GenericDownloadScript {
    webServerHost ATTRIB WebHost;
    webServerPort ATTRIB WebPort;
    srcfile "/~bikash/guestbook.war";
    installLocation extends concat {
        jbossHome ATTRIB JBossHome;
        -- "/server/default/deploy/";
    }
    destfile "guestbook.war";
   }
}
```

We assume that the guest book application is available as an archive. we need to download it to appropriate location inside the jboss directory in order to deploy it. This component uses the GenericDownloadScript component and defines the appropriate attributes in order to perform the task.

Integration In order to automate the whole process of deploying the two tier application, jboss, postgres as well as the servelt application needs to be deployed. The following component description perform these steps.

```
#include "postgresApp.sf"
#include "jbossApp.sf"
#include "servletApp.sf"
twotier extends Sequence {
    JBossHome "/home/bikash/Automation/Install/jboss-3.2.2/";
    actions extends LAZY {
      ServletApp extends servletApp;
      OtherApp extends Compound {
        jboss extends jbossApp;
        pgres extends postgresApp;
      }
    }
}
sfConfig extends twotier;
```

The servlet deployment is done initially. Once it is complete, the deployment of jbossApp and postgresApp are performed in parallel through the *Compound* workflow in smartfrog. This way, components can be nested together and a graph can be instantiated.

Using smartfrog for deploying any applications requires a description of the component in the smart frog language, an implementation of the component and its life cycle management functionalities through a java program or wrapper. The code and configuration file can be l0ocated locally or remotely. Setting up database for use, setting up environment variable for all the users so that the newly installed software can be used by everyone etc, should be performed through appropriate components.

2.3 Discussion

We were able to completely automate the process of OS installation and 3-tier application installation on small number of machines (<5) in the data center. Detailed quantitative analysis of scalability of the system was not possible, however, since the system takes decisions based on policies, and since resource allocation system is coupled with the installation and deployment system, hence, if system load exceeds beyond acceptable limit (as defined by policy), new servers can be installed and deployed. Separating the various system components, defining clear interfaces between the various components through grid services, policy based decision making, having multiple server for load balancing, and automation of the whole process leads to a scalable solution to the problem of provisioning complete software stack in a grid setting.

3 Related Work

We are not aware of any system which performs the automatic provisioning of the complete software stack in a grid environment. However, separate solutions exist which perform the OS installation, application installation and deployment. NPACI Rocks [4], Kickstart [10], Jumpstart [9] are some of the tools for automating the OS installation. SmartFrog [7], LCFG [3] and cfEngine [5] are some of the tools used for configuration management of a node. cfEngine can be used to perform specific configuration tasks such as file operations(linking, setting permissions), editing text files, bringing up network interfaces, and monitoring important files for changes. However, cfEngine doesn't provide support for OS installation or e-commerce application deployment. SmartFrog provides a general framework for deployment and life-cycle management of applications but it doesn't provide support for OS installation. Moreover, our implementation uses specialized SmartFrog component for deploying e-commerce applications. LCFG provides support for automating the OS installation but as such, it is not suitable for provisioning e-commerce applications in grid environment. In summary, none of these existing tools is suited alone for the provisioning of the complete software stack in a grid. We have integrated SmartFrog and kickstart with intelligence in the middleware, which leads to a more scalable solution. We have also identified the various components which can be implemented as grid services.

4 Conclusion

We have demonstrated the automation of installation, configuration, and deployment of software stack in a data center environment. We have shown the usefulness of our system by installing linux OS and 3-tier application as well as deploying the 3-tier application in a data center. We used kickstart with PXE for performing the automated OS installation on our target nodes. We also used smartfrog in order to install and deploy applications on the target nodes. We have developed some generic smartfrog components which can be used to start, stop, and perform limited life-cycle management of any application which doesn't require any user inputs during start up.

We have been successful in automating the provisioning of complete software stack in a grid environment. We have come up with a system architecture for performing the installation and deployment of OS and applications in a grid environment and identified the interfaces between the various modules. We demonstrated its usefulness through a concrete implementation. We have mainly focussed on the scalability aspect of the system, however, security, remote monitoring, load sharing of the various server components are some of the issues that can be looked into in future.

References

[1] JDBC 3.0 specification. *Available online from http://java.sun.com*.

[2] Postgresql database server. *http://sourceforge.net/projects/pgsql*.

[3] Paul Anderson. Towards a hi-level machine configuration system. *In Proceedings of the 8th Large Installations System Administration (LISA) Conference*, pages 19–26, October 1994.

[4] G. Bruno, P.M. Papadopoulos, and M.J. Katz. Tools and techniques for easily deploying manageable linux clusters. *3rd IEEE International Conference on Cluster Computing (CLUSTER)*, 14:258–270, October 2001.

[5] Mark Burgess. Cfengine: a site configuration engine. *USENIX Computing Systems*, 1995.

[6] Intel Corporation. Preboot execution environment (pxe) specification. *Available online from http://www.intel.com/labs/manage/wfm/wfmspecs.htm*, September 1999.

[7] P. Goldsack. Smartfrog: Configuration, ignition and management of distributed applications. *http://www-uk.hpl.hp.com/smartfrog*, 2003.

[8] JBoss Group. Jboss. *http://www.jboss.org*.

[9] Sun Microsystems Inc. Solaris 8 advanced installation guide. *Available online from http://docs.sun.com*, 2002.

[10] Red Hat Incorporated. Red hat linux 7.3: The official red hat linux customization guide. *Available online from http://www.redhat.com*, 2002.

[11] Vanish Talwar, Bikash Agarwalla, Sujoy Basu, Raj Kumar, and Klara Nahrstdt. A resource allocation architecture with support for interactive sessions in utility grids. *In Proceedings of 4th International Symposium on Cluster Computing and the Grid (CCGRID'04)*, April 2004, (*to appear as poster*).