Leveraging Standard Core Technologies to Programmatically Build Linux Cluster Appliances

Mason J. Katz, Philip M. Papadopoulos, Greg Bruno
The San Diego Supercomputer Center
University of California San Diego
La Jolla, CA 92093-0505
[mjk,phil,bruno]@sdsc.edu
http://rocks.npaci.edu

Abstract

Clusters have made the jump from lab prototypes to full-fledged production computing platforms. The number, variety, and specialized configurations of these machines are increasing dramatically with 32 – 128 node clusters being commonplace in science labs. The evolving nature of the platform is to target generic PC hardware to specialized functions such as login, compute, web server, file server, and a visualization engine. This is the logical extension to the standard login/compute dichotomy of traditional Beowulf clusters. Clearly, these specialized nodes (henceforth "cluster appliances") share an immense amount of common configuration and software. What is lacking in many clustering toolkits is the ability to share configuration across appliances and specific hardware (where it should be shared) and differentiate only where needed. In the NPACI Rocks cluster distribution, we have developed a configuration infrastructure with well-defined inheritance properties that leverages and builds on de facto standards including: XML (with standard parsers), RedHat Kickstart, HTTP transport, CGI, SQL databases, and graph constructs to easily define cluster appliances. Our approach neither resorts to replication of configuration files nor does it require building a "golden" image reference. By relying on this descriptive and programmatic infrastructure and carefully demarking configuration information from the software packages (which is a bit delivery mechanism), we can easily handle the heterogeneity of appliances, easily deal with small hardware differences among particular instances of appliances (such as IDE vs. SCSI), and support large hardware differences (like x86 vs. IA64) with the same infrastructure. Our mechanism is easily extended to other descriptive infrastructures (such as Solaris Jumpstart as a backend target) and has been proven on over a 100 clusters (with significant hardware and configuration differences among these clusters).

1. Introduction

Within the last decade, the Network of Workstations [5] and Beowulf [11] models of COTS clustering have made substantial gains in the HPC marketplace previously held by single vendor supercomputers. This trend shows no sign of slowing, and Linux-based clusters are paving the way on the Grid in the form of efforts such as NSF's Distributed Terascale Facility [1]. As commodity clusters scale out and challenge traditional big-iron supercomputers, a new model of software configuration is needed to accommodate the heterogeneity that creeps into the system as components fail, subsystems are upgraded, and application requirements change. Existing techniques of system imaging and description-based installation (e.g., RedHat's Kickstart) fail to accommodate the innate heterogeneity of large-scale clusters and require forking either along the lines of disk and file system images or Kickstart files. Although the former is much harder to manage in terms of sheer disk space requirements, the later isn't much better. A primary goal of managing the software configuration of a cluster should be to avoid replication of components and configuration. Currently, neither system imaging nor Kickstart possess this property.

Our previous version of NPACI Rocks clustering software (version 2.0) leveraged RedHat's Kickstart utility to manage the software and configuration of all nodes. We enhanced Kickstart to use the C pre-processor as a macro language and extracted site configuration variables from an SQL database. Using these tools we were able to use a single source file to describe multiple node configurations, but we still required two separate source files – one for describing the frontend node of the cluster and another for compute

nodes. Although this successfully decoupled node-specific configuration from generic node configuration, there was still replication in the Kickstart files between compute node and frontend configurations. At the time, this was a reasonable compromise as we felt a cluster would only have two node types (frontend and compute). We've since realized a cluster has many node types (hereafter referred to as "appliance types" or "appliances"). Although our implementation had several desirable features, it ultimately proved too awkward and rife with subtleties injected from the *ad hoc* nature of the design.

Our solution is to decompose the configuration of a cluster into "appliances" where each appliance is composed of several small single-purpose configuration modules. Further, all site- and machine-specific information is managed in an SQL database to allow these configuration modules to be shared between cluster nodes and cluster sites. For example, a single module is used to describe the software components and configuration of the `ssh` service. Cluster appliance types which require `ssh` are built with this module. A single object-oriented framework is used to build the module hierarchy, resulting in multiple cluster appliances constructed from the same core software and configuration description. This framework is composed of XML files and a Python engine to convert component descriptions of an appliance into a RedHat compliant Kickstart file.

2. Related Work

- **SystemImager** - SystemImager [8] is a Linux software installation tool that stores a system's operating environment (e.g., OS, tools, libraries and configuration) at the *file-level*, that is, the installation server holds an image of the entire directory structure for a configuration.

  This is opposed to *bit-level* installation tools that store bit images (e.g., a copy of the bits on a disk, usually the output of `dd`). A file-level imager has advantages over bit-level imagers in that small changes to files only require the changed files to be transferred to the imaged nodes rather than the entire disk image. In SystemImager, local node software configuration comes in the form of leveraging DHCP, adding values to a machine specific `local.cfg` file, or by augmenting the installation script (*autoinstall script*).

  If a cluster has heterogeneous hardware, that is, if all the nodes don't have the same installed hardware, SystemImager supports each hardware platform by storing a unique image of the directory structure for each hardware type. Cluster node configuration for a specific hardware configuration is managed by first installing one node in the cluster, logging into the node and configuring the software by hand, then taking a snapshot of the node (i.e., creating a *master node*). Conversely, in description-based methods, cluster node configuration is specified *a priori*. Hardware and software configuration can be supported by providing unique descriptions for each hardware/configuration type or by engineering a method that examines nodes and programmatically adapts the local node configuration based on the node's unique characteristics.

- **Cfengine** - Cfengine [6, 7] is a policy-based configuration management tool that can configure UNIX or NT hosts. After the initial operating environment is installed by hand or another tool (cfengine doesn't install the base environment), cfengine is used to instantiate the initial configuration of a host and then keep that configuration consistent by consulting a central policy file. The central policy file is written in a cfengine-specific configuration language (which resembles makefile syntax) that allows an administrator to define the configuration for all hosts within an administration domain. Each cfengine-enabled host consults this file to keep its configuration current.

  To deal with hardware and software heterogeneity, cfengine defines *classes* to delineate unique characteristics.

- ***x*-Kernel** - The *x*-kernel [9] is a network operating system which decomposes traditionally monolithic network stacks into simple single-purpose micro-protocols. In addition to these micro-protocols, a single graph file describes the relationships of the protocols in the context of a network stack. It is this decomposition of network stacks into a bag of protocols and a single graph file to instantiate a specific network service that made the *x*-kernel an attractive platform for exploring ideas within network systems research. The authors describe the novelty of the *x*-kernel as an attempt to apply pragmatic software engineering methods in the form of program encapsulation, re-usability, and composibility to the problem domain of network stacks[10].

  Along these lines, our work has applied simple software engineering techniques to the problem domain of system software configuration. Further the distinction between a bag of micro-protocols and a single description of the protocol framework is analogous to our concept of single purpose configuration nodes and a configuration graph.

- **RedHat's Kickstart** - RedHat has written a sophisticated, customizable script which automates package installation from RedHat distributions called *Kickstart* [3]. Nodes installed in this manner are driven by a

```
url --url http://10.1.1.1/install/i386
zerombr yes
clearpart --all
part / --size 4096
lang en_US
keyboard us
mouse genericps/2
timezone --utc GMT
skipx
install
reboot

%packages
@Base
pdksh

%post
cat > /etc/motd  << 'EOF'
Kickstarted on `date`
EOF
```

Figure 1. Sample RedHat Kickstart file.

user-created configuration file that essentially contains the answers to all the questions posed by a standard interactive installation. Figure 1 presents a sample Kickstart file. The file has three sections: *command*, *package*, and *post*. The *command* section contains almost all the answers posed by an interactive RedHat installation (e.g., location of the distribution, disk partitioning parameters and language support). The *packages* section lists the names of RedHat Packages (RPMs) to be installed on the machine. Finally, the *post* section contains scripts which are run during the installation to further configure installed packages.

While a Kickstart file is a text-based description of all the software packages and software configuration to be deployed on a node, it is static and monolithic. At best, this requires separate Kickstart files for each host type. At worst, this requires a separate file for each host. The success of Kickstart is in providing a *de facto* standard for installing software and performing the system probing required to install and configure the correct device drivers on a per machine basis.

- **LCFG** - A project that is closely aligned with the methods described in this paper is the LCFG project [4]. Anderson and Scobie have designed a system around the observations that node configurations change often, that configuration information needs to be stored independently of the host systems and that handling automated installation and configuration for heterogeneous clients (from large servers to laptops) in an evolving environment using a single method is desirable.

  Both LCFG and our method use a collection of source files, with each source file dedicated to configuring exactly one service - to generate a tailored configuration file for a host. Both rely on a central database to assist in the application of node-specific information to the resulting configuration file. Both also have a notion of *inheritance* to leverage source file reuse for similar machine types.

  LCFG differs from our method in that inheritance is supported in LCFG through file inclusion (e.g., `#include`). Our first implementation also used file inclusion, but we evolved this to a graph structure which has proven to be simpler and more powerful due to the structured nature of the resultant Kickstart configuration file. LCFG also employs a proprietary configuration language for their source files. A custom *profile* compiler is used to combine the source files into single XML profile. Our method uses XML as the structure for the source files. LCFG doesn't use Kickstart to install the operating environment. Rather it uses its own boot environment to configure the machine (e.g., to detect the hardware, partition the disk, install RPMs). We leverage Kickstart's extensive hardware probing and configuration mechanisms in order to support a wide-range of hardware platforms without having to develop code.

3. Component based configuration

The key pieces missing from Kickstart are a macro language and a framework for code re-use. The former can be solved with the C pre-processor `#define` directives. The later can be solved by breaking Kickstart files into modules and using the C pre-processor to allow modules to include others to build complexity. Although this will minimally solve the problem, this implementation proves unwieldy and results description files which are difficult to read and maintain.

The success of our initial `cpp` approach was in decomposing the monolithic Kickstart file into dozens of small modules of dedicated functionality. For example, a single file contained the Kickstart commands for installing and configuring a web server. The failure was the level of effort required to support this structure. There was a need for a structured, standard representation of these modules. This same problem is evident in C software which is structured in an object-oriented manner but must invent its own mechanisms to support object-oriented concepts (as opposed to writing software in a language that directly supports object-oriented methodologies, like C++). In this vein, XML offers a structured representation for the Kickstart modules. In addition, XML opens the door to *de facto* standards for parsing data, allowing us to focus on engineering appropriate software components, and not on parsing.

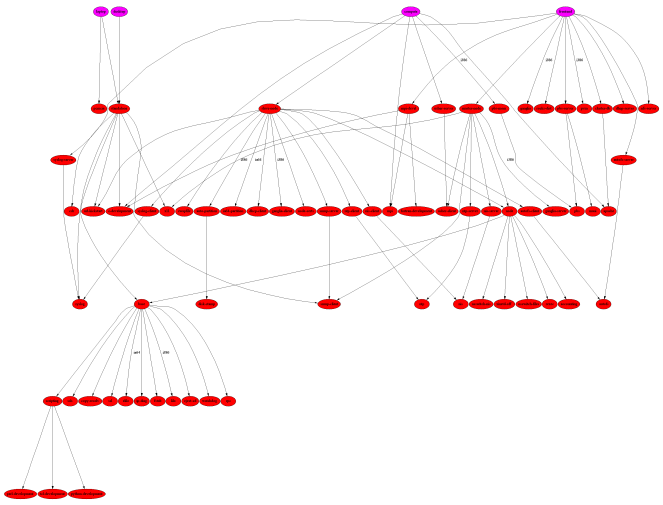
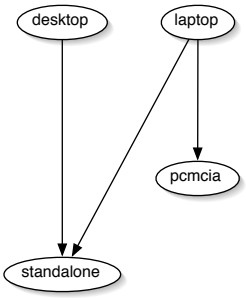
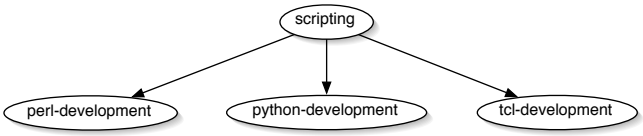


Figure 2. Kickstart Graph.




```
<graph>
  <edge from="desktop" to="standalone"/>
  <edge from="laptop"  to="standalone"/>
  <edge from="laptop"  to="pcmcia"/>
</graph>
```

Figure 3. Specifying minor configuration differences with inheritance.




```
<graph>
  <edge from="scripting" to="perl-development"/>
  <edge from="scripting" to="python-development"/>
  <edge from="scripting" to="tcl-development"/>
</graph>
```

Figure 4. Building compositional configuration objects.

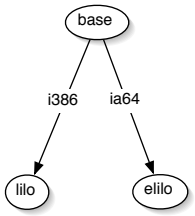
Once the functionality of a system is broken into small single-purpose modules, a framework describing the inheritance model is used to derive the full functionality of complete systems, each of which shares common base configuration. Figure 2 is a representation of such a framework which describes the configuration of appliances in a Rocks cluster. The framework is a directed graph – each vertex represents the configuration for a specific service (software package(s), service configuration, local machine configuration, etc.) Relationships between services are represented with edges. At the top of the graph there are four vertexes which indicate the configuration of a “laptop”, “desktop”, “frontend”, and “compute” cluster appliance. The full description an appliance is built by traversing the graph, starting with the appliance, thereby inheriting all the functionality which is shared across other appliances.

The framework is composed of a single graph file and nearly a hundred configuration modules. The graph file specifies the framework hierarchy, and the configuration modules (also called ”nodes”) each specify a distinct piece of system configuration.

3.1. Graphs

The traditional object-oriented model of inheritance requires objects to specify their ancestor. In C++ this is explicitly done in the class declaration of an object. However, one of the most difficult problems in object-oriented design is the specification of the inheritance model for all the objects. This is a difficult problem for even small class libraries. In the case of Kickstart nodes, we have nearly one hundred unique nodes in our framework. As cluster users invent new ideas for cluster appliance types, a fixed framework of these nodes becomes a problem. If the relationship between two nodes in the configuration graph is incorrect for a new appliance type, the power of an object-oriented framework is lost.

To solve the problem of finding the correct class hierarchy, we avoid it altogether. Individual nodes do not specify their ancestor nodes. Rather the inheritance model is specified out-of-band in a separate XML graph file. This allows for a default framework which works for the appliances we need while allowing our users to invent their own frameworks. Our goal has been to provide the structure to effectively build computational clusters in the style we currently recommend, while keeping the door wide open for radically different clustering (or non-clustering) configuration ideas. Along these lines, we have designed a system which invites experimentation while still providing a simple and complete default HPC cluster configuration.




```
<graph>
  <edge from="base" to="lilo"  arch="i386"/>
  <edge from="base" to="elilo" arch="ia64"/>
</graph>
```

Figure 5. Handling architecture differences with conditional inheritance.

Figure 3 shows a small section of the graph from Figure 2. In this example, the "desktop" and "laptop" appliance types both inherit from a "standalone", yet a singular difference of "laptop" inheriting from "pcmcia" insures support for the PCMCIA bus which desktops do not have. Likewise, the XML representation is a subset of the XML used to describe Figure 2. In addition to building appliances, nodes can be used to collect related functionality into a single entity. Figure 4 shows a single "scripting" node built by inheriting from nodes for all the common UNIX scripting languages.

In addition to simple inheritance, the graph supports conditional inheritance based on a machine's system architecture. When a Kickstart file is built for a machine, the CPU architecture is specified. By annotating the edges of the graph with an "arch" attribute, a single framework supports the configuration of cluster appliance types for multiple architectures. Figure 5 shows how the "base" node in the graph inherits either "lilo" or "elilo" depending on whether the target machine is an x86 or Itanium. This feature of the graph embraces, rather than rejects, heterogeneous architectures thereby allowing the system architect to work at the appropriate level of abstraction. We've found this to be one of the more interesting features of the system. By exploiting this feature, a single core set of software has been used to support x86 and IA64 releases of Rocks.

3.2. Nodes

Each node in the graph corresponds to a single XML file. The tags correspond directly to RedHat Kickstart keywords. An XML representation is used to provide a simple and standard grammar for the file. Further, XML provides simple macros in the form of *entities* and allows us to define our own more powerful macros.

Figure 6 shows the XML file for an "ssh" node in the

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE kickstart SYSTEM "dtds/node.dtd"
[<!ENTITY ssh "openssh">]>
<kickstart>

  <package>&ssh;</package>
  <package>&ssh;-clients</package>
  <package>&ssh;-server</package>
  <package>&ssh;-askpass</package>

  <!-- Required for X11 Forwarding -->
  <package>XFree86</package>
  <package>XFree86-libs</package>

<post>
<!-- default client setup -->
cat &gt; /etc/ssh/ssh_config &lt;&lt; 'EOF'
Host *
        CheckHostIP             no
        ForwardX11              yes
        ForwardAgent            yes
        StrictHostKeyChecking   no
        UsePrivilegedPort       no
        FallBackToRsh           no
        Protocol                1,2
EOF
</post>

</kickstart>
```

Figure 6. The ssh.xml node includes the ssh packages and configures the service in the Kickstart post section.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE kickstart SYSTEM "dtds/node.dtd">
<kickstart>
  <main>
    <lang><var name="Kickstart_Lang"/></lang>
    <keyboard><var name="Kickstart_Keyboard"/></keyboard>
    <mouse><var name="Kickstart_Mouse"/></mouse>
    <timezone><var name="Kickstart_Timezone"/></timezone>
    <rootpw>--iscrypted <var name="RootPassword"/></rootpw>
    <install/>
    <reboot/>
  </main>
</kickstart>
```

Figure 7. The base.xml node configures the main section of the Kickstart file

graph. This node has a single purpose - to describe the packages and configuration associated with the installation of the ssh service and client on a machine. The *package* and *post* XML tags map directly to Kickstart keywords. The "ssh" node also makes use of an XML entity as a simple macro to set the *ssh* prefix to *openssh* (as opposed to *ssh*, which RedHat supported prior to their 7.1 release).

Although simple macros are quite powerful, we needed a method of isolating site and individual cluster host state from the generic configuration description. The configuration can be thought of as the program which is used to configure a set of software. This program has state, which represents a single instantiation of a cluster appliance. Our approach is to store this state in an SQL database and allow the XML configuration code to reference these variables. Figure 7 shows the XML for a "base" node in the graph. This node configures much of the *command* section of a Kickstart file, such as the language, timezone, and keyboard. Rather than specifying this state in the XML file, the *var* tag is used to reference it. Unfortunately the current semantics of entities in XML proved insufficient to allow them to be used for this purpose. Hence, the additional *var* tag was invented.

In cases where *var* tags cannot completely represent the machine state variables, an *eval* tag can be used. *Eval* statements are used to evaluate arbitrary expressions with the resultant values replacing the *eval* section. The *eval* tag should not be viewed merely as an escape hatch to the confinements of the XML framework, but rather as a method of making a Turing machine available for computing machine-specific configuration. We've used this feature to intelligently allow a host to determine its own optimal disk partitioning scheme, and to support complex SQL queries that did not map to the simple key/value pairs supported by the *var* tag.

4. System Architecture

4.1. Components

- **Anaconda** - Anaconda is the name of the UNIX process for RedHat's Kickstart client. Each machine in the cluster runs this client from a floppy, cdrom, hard disk, or PXE boot. The Anaconda process requests a Kickstart file (either locally from the installation media or over the network, for example HTTP or NFS), parses the keywords from the file and executes the appropriate commands to completely configure the software installation of the machine. Once Anaconda completes, the machine reboots and becomes a functioning member of the cluster.

- **CGI** - Rocks uses a web CGI script to serve Kickstart files to Anaconda clients. When a node installs, it requests its Kickstart file using HTTP. The node constructs the URL by combining information from the DHCP response and node-specific information (e.g., hard disk name(s) and architecture type). An example URL for a x86-based node with two SCSI disks is:

  ```
  http://frontend-0/install/kickstart.cgi?
  devnames=sda,sdb&arch=i386
  ```

  The CGI script coordinates the creation of a Kickstart file by extracting node-specific fields from the query component of the URL, querying the SQL database, and passing these values into subsequent programs - `KPP` (generates a monolithic XML file from XML-based components) and `KGen` (transforms the monolithic XML file into a valid RedHat Kickstart file). The CGI script takes the output of `KGen` and streams it back to the installing node.

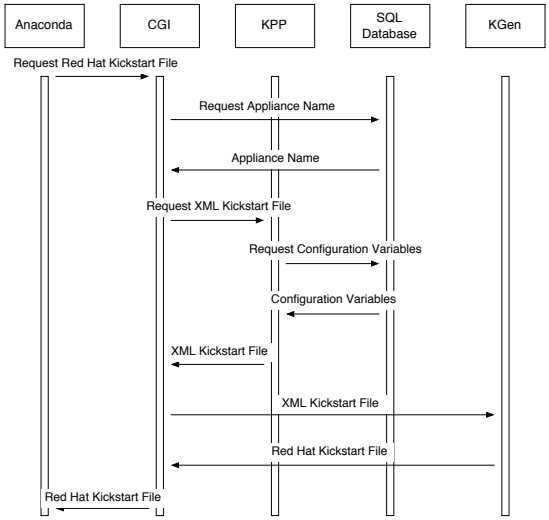


Figure 8. Kickstart space-time diagram.

- **SQL Database** - The configuration database stores information about the cluster as a whole and information about specific machines and groups of machines.

- **KPP** - The Kickstart Pre-Processor traverses the configuration graph, requests machine state from the SQL configuration database and builds a single monolithic XML-based Kickstart file for a specific cluster node.

- **KGen** - The Kickstart Generator transforms an XML Kickstart file into RedHat Kickstart syntax. This additional step exists to allow future formats to be used. For example, it should be possible for another generator to produce Solaris JumpStart files.

4.2. Process

Figure 8 illustrates the process of a machine requesting and receiving its Kickstart file. First, the Anaconda process issues an HTTP request of the Kickstart file, which triggers the CGI script. This script interrogates the configuration database to determine which appliance type should be instantiated on the machine and requests an XML Kickstart file for the specific machine and appliance type from the Kickstart Pre-Processor. [1] KPP interrogates the configuration database for machine-specific configuration variables to resolve the XML *var* tags in the node file, then graph traversal starts at the appropriate appliance node and KPP builds a single machine specific XML Kickstart file which is returned to the requesting CGI script. Finally, the CGI script invokes the Kickstart Generator to convert the XML syntax into native RedHat Kickstart supported syntax.

The end result is that the Anaconda process is unaware that a dynamic system generated the Kickstart file and behaves as if it had received a standard static Kickstart file. This simple fact allows us to leverage all of RedHat's efforts, while adding the necessary power to program the behavior of a cluster.

5. Future Work

The visualization of the configuration graph has proved to be one of the most compelling pieces of this work.[2] The visualization has shown us that the model of a graph makes system configuration highly accessible to users and should be the interface for system configuration. We would like to leverage existing drawing packages to draw the relationships between nodes and make clicking on an individual node bring up an editor window for modifying the corresponding XML file. In this fashion, the metaphor of the graph would never be hidden from the system architect.

We also want to apply this configuration graph to other problems. Currently only the initial Kickstart software deployment is described in the framework. The same framework should be able to describe system auditing, verification, and dynamic system configuration. In this manner the graph can be applied to configuring intrusion detection tools (e.g., Tripwire [2]) and cfengine-like tools.

Finally, we are interested in applying this to other operating systems which support description-based software installation. Solaris JumpStart is the obvious target, but future releases of Windows may also fit into the model.

[1] This architecture also supports the client machine telling the CGI what appliance type it wishes to become. We envision exploiting this feature to build clusters of machines which autonomously re-tool themselves as different appliance types to meet the dynamic demands on the cluster.

[2] Graph images are drawn using the `dot` command from the graphviz package from AT&T. When invoked without an appliance type, KPP outputs a dot-compliant file rather than an XML Kickstart file.

6. Conclusions

Description-based software deployment has been a dramatic leap forward over past bit-level and filesystem-level cloning. Description-based installation provides the appropriate level of abstraction to support hardware heterogeneity and makes the explicit distinction between two orthogonal concepts – software component names and software component versions.

Although RedHat Kickstart provides these benefits, it has many of the pitfalls of traditional system cloning in that it does not provide the necessary programmability to avoid code replication, which is often a pitfall of software engineering efforts. By applying standard software engineering concepts in the form of a simplified object-oriented framework, we have added programmability to Kickstart to build multiple variants of cluster appliances on heterogeneous architectures without replicating common appliance configurations.

References

[1] Distributed Terascale Facility to Commence with $53 Million NSF Award. http://www.nsf.gov/od/lpa/news/press/01/pr0167.htm.

[2] Tripwire. http://www.tripwire.org/.

[3] Red hat linux 7.1: The official red hat linux customization guide. http://www.redhat.com/docs/manuals, 2000.

[4] P. Anderson and A. Scobie. LCFG: The next generation. In *UKUUG Winter Conference*, 2002.

[5] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for networks of workstations: NOW. *IEEE Micro*, Feb. 1995.

[6] M. Burgess. Cfengine a site configuration engine. volume 8. USENIX Computing Systems, 1995.

[7] M. Burgess. Recent developments in cfengine. The Hague, 2001. Unix.nl Conference.

[8] B. E. Finley. VA SystemImager. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, Oct. 2000.

[9] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[10] S. O'Malley and L. Peterson. A dynamic network architecture, 1992.

[11] T. L. Sterling, J. Salmon, D. J. Becker, Savarese, and D. F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, 1999.