Clusters as Large-Scale Development Facilities[1]

Rémy Evard, Narayan Desai, John-Paul Navarro, and Daniel Nurmi
{evard, desai, navarro, nurmi}@mcs.anl.gov

Abstract

In this paper, we describe the use of a cluster as a generalized facility for development. A development facility is a system used primarily for testing and development activities while being operated reliably for many users. We are in the midst of a project to build and operate a large-scale development facility. We discuss our motivation for using clusters in this way and compare the model to a classic computing facility. We describe our experiences and findings from the first phase of this project. Many of these observations are relevant to the design of standard clusters and to future development facilities.

1 Background

The objective of Argonne National Laboratory's Chiba City Project [1] is to provide a computing platform for development and testing of large-scale high-performance computing software while carrying out research in systems software (e.g., the software needed to manage and operate the systems and to support applications). We have two primary motivations for this work:

- Scalability is a fundamental goal of high-performance computing. Much research during the past decade has demonstrated that the primary barrier to achieving systems scalability is scalability of systems software.
- Researchers investigating paths to petaflops-capable systems in the early and mid-1990s identified multiple possible hardware technology paths to petascale performance. Each of these hardware paths, regardless of the technology base, had one thing in common: the need for increasing degrees of concurrency in future systems. Future systems are likely to have hundreds of thousands to millions of individual components.

In essence, future high-end systems will be substantially larger scale than today's systems, perhaps by three or more orders of magnitude. System software, libraries, and applications must be able to operate effectively at this scale.

The explosive growth of commodity-based clusters has reinforced these expectations. Many institutions have demonstrated that one can effectively build very large systems out of small and cheap individual components. As processor technology continues to shrink in size and cost, to increase in capability, and to become specialized, clusters will continue to grow in size and capability. However, the scalability of systems software has not kept pace with the growth of clusters. It is still true that one of the biggest challenges in the cluster computing community is the development of system software that scales reliably.

One of the barriers to the development of such systems software is that facilities that support developing and testing at scale are rare. The vast majority of large computers in existence are dedicated to computational simulation, not to development and testing. Developers only have limited access and time on these systems, and any kind of development that might destabilize the

[1] This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Dept. of Energy, under Contract W-31-109-ENG-38.

system (such as file system or scheduler changes) must be done with extreme care, if at all. The standard batch-queue model of supercomputer scheduling is not conducive to development or exploration of ideas, both of which tend to require interaction. This situation substantially limits the amount of research and development that can be put into scalability issues, which in turn tends to cause people to focus their effort in other directions.

The first Chiba City cluster was installed in 1999 as a facility for large-scale development and testing in order to help address the lack of testbeds for system software developers and to promote research into scalability issues in all areas of high-performance computing. Demand for the development capabilities of system is increasing strongly. We are now in the process of designing the next generation of the facility. In this paper, we describe our experiences and observations from the first phase of this project.

2 A Development and Testbed Facility

We designed and built Chiba City by incorporating our own experience in supporting a decade of research activities on a wide range of parallel platforms with the designs of other clusters and input from potential early users of the system. The capabilities of the system have changed over time as we have come to better understand the requirements of the testbed community. Here we describe the initial capabilities and features of the system that were focused on testbed activities.

2.1 System Components

To first order, Chiba City looks very much like a standard cluster used for computational science e.g. a BEOWULF system [2]. (See Figure One.) The components of the system can be categorized as follows:

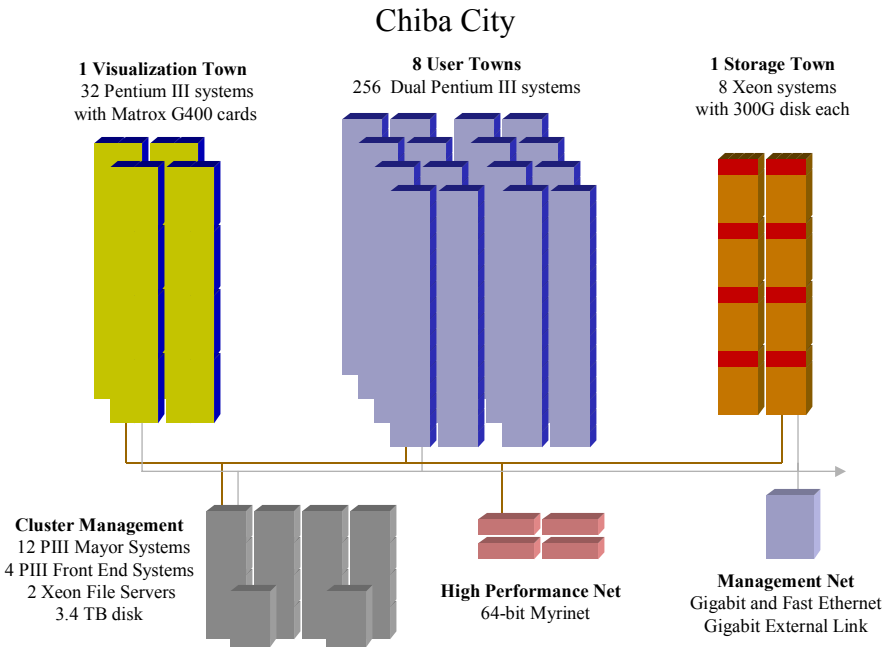

Figure 1 – Chiba City Diagram

- **User nodes** (256). These nodes run the user's "jobs".
- **Login nodes** (2). These are the front-end nodes, with a full Linux computing environment. This is where "jobs" are submitted to a scheduler.
- **Management nodes** (11). These systems manage the user nodes, but are not visible to the users.
- **Storage nodes** (8). These systems provide file system service to the user nodes and are accessible to users carrying out I/O experiments.
- **Visualization nodes** (32). These systems support graphics and visualization experiments.
- **Myrinet**.
- **Switched fast/gigabit Ethernet**.

A more detailed description of the Chiba City environment is in [1]. In practice this system organization has worked fairly well.

2.2 The Management System

The management system has been essential to the testbed activities. The cluster was designed around the concept of having a robust and reliable management system that could be used to manage and control the rest of the cluster.

The physical components of the management system include:

- The management nodes mentioned above. These are organized hierarchically; there is one primary management node that controls all of the other management nodes. Each of these in turn control and monitor up to thirty-two user nodes.
- A serial console infrastructure, allowing access to all system consoles via the network. The management systems monitor the consoles for status and occasionally issue commands over the consoles.
- A simple power control infrastructure that allows administrators to power cycle any hardware in the system.

Management of the system requires a suite of software tools. Some of these are standard systems administration and cluster tools from the open source community, while others were developed in house. The most important of these in relation to this paper is the "ChibaDB" – a database of information about the cluster, including node hardware information and node software configuration. When a node is rebooted or rebuilt (as described in section 4.1), the management system uses the database to determine which operating system to install on that node.

2.3 System Usage

While the cluster was to be dedicated to supporting testbed activities, we also intended to support computational science when cycles on the system were available. Priorities for cluster usage were set in this order:

1. Computer science researchers and developers.
2. Computational scientists who were willing to act as testers of the potentially unstable system software.
3. Computational scientists testing code at large scale.
4. Other potential users.

Under normal circumstances, a batch scheduler mediates access to the cluster, just as if it were a standard computational cluster.

2.4 Initial Capabilities

Based on the requirements identified by the initial users of the system, we designed a system that gave users the following abilities:

- **On-Demand Access**: the ability to run programs on the system without waiting in the batch queue. We have supported this in a number of ways, some of which have changed over time.
- **Interactivity**: the ability to interact with a program or the OS on a user node. This is a fairly standard feature on most clusters, as it is important for debugging.
- **Root Access**: having root access on one or more user nodes of a system.
- **Dynamic OS**: the ability to install a non-standard operating system, or variant of a standard one, on one or more user nodes.

We were able to grant root access and dynamic operating systems on the user nodes by building a cluster management infrastructure that could reboot and install an operating system on any user node in the system. Thus, after a user completed using a set of nodes in this way, we could "clean up" and re-install the standard OS before allocating those nodes to the next user.

This set of features was sufficient for the early use of the system. However, as our understanding of the needs of testbed users has grown, we have modified and extended the basic requirements list accordingly.

2.5 Relation to Computational Clusters

The standard usage of a testbed cluster is similar to the usage of a computational cluster:

- Users request resource allocations and run jobs on the resulting nodes.
- Job runs consist of the following steps:
    - The user supplies the application and input data.
    - The user then specifies the way that the application is executed.
    - Finally, the output data produced by the application is returned to the user.

While this high level description applies to both computational and testbed clusters, the specifics in both cases are substantially different. We illustrate this using the case of a testbed user who is developing some aspect of an operating system and therefore installs a custom image as part of their activity:

	Computational User	**Testbed User**
Desired resource	Some number of nodes with a standard (and usually minimal) OS.	Some number of nodes with a custom and user-specified OS.
Application	A user-mode application.	An OS with custom features.
Input	Typically data sets or input parameters.	Test cases to be executed.
Output	Typically some form of numerical result.	Results of tests, both qualitative and quantitative.

In essence this means that the basic job model for both uses is similar in character. (Indeed, we support both types of jobs on the system, often simultaneously.) However, the specific goals of a testbed user's job are often quite different, which in turn means that the usage patterns within that job are different.

3 Testbed Usage Characterization

In the first days of the system, testbed usage was largely characterized by users who needed interactive access to a large cluster for short periods of time. For example, scientists used the cluster to test systems software that launched jobs. In this case, which is typical of much of this type of development, the scientists would need to use the entire system interactively, but only for moments at a time.

Interestingly, the usage of the system has changed over time, both as the system has become more capable and as the user community of Chiba City has grown. Recently, we have had quite a number of different OS and system tool developers on the system who need to install their own operating systems on as much of the cluster as possible. After pushing out and configuring the installation, they usually run a series of tests that might take hours or days.

We can classify testbed usage based on two metrics:

- Degree of scalability. This describes the degree to which the specific project is focused on developing and testing at large scale, or carrying out research into scalability issues.
- Degree of system impact. This describes the project's ability to operate in the standard environment. We have categorized these as "computational usage", "basic development", "system development", and "extreme development", each of which is described below.

These two issues, scalability and impact, go hand in hand. While the testbed can support high-impact development on a single node, most high-impact testbed users are also interested in testing scalability issues. Therefore, while we have found it interesting to note which of our users operate at large-scale, we have not found it particularly useful to differentiate between them based on scalability because most of the testbed users eventually want to use the entire system.

In the following sections, we profile these broad categories of cluster users and describe the augmented functionality they require in order to effectively use testbed clusters. They are described in order of increasing degree of system impact.

3.1 Computational Usage

The first type of user is a standard application user on a computational cluster. In most of these cases, the user has a mature application that they want to run for some period of time. The operating system running on the nodes usually doesn't matter as long as the application can be recompiled for the target system. No enhanced privileges of any sort are required on any portion of the system. A computational user may place significant demand on the I/O system of the cluster.

The intent of a computational application user is typically to generate set of numerical results.

3.2 Basic Development

The second type of user, which is quite common on Chiba City, is the basic development user. A good example of this type of user is a system library developer such as a numerical library or a communications library.

In general, these types of programmers are interested in the scalability and performance of their code. They have the following requirements, some of which were noted above:

- **On-demand access**. Waiting in a queue when actively developing can severely limit the effectiveness of a development session. These users like to be in the "code/compile/test" loop that is common for development on unscheduled systems. In order to address this issue, we took three steps:

- Thirty-two of the nodes on the system are "unscheduled", i.e. the scheduler does not control access to them. They are available to all users at all times, and are specifically meant to be used as an on-demand testing area.
- The scheduler policy was arranged to allow only very short-running jobs during certain business hours. In theory, this would allow the quick jobs that characterize this kind of development to migrate to the front of the queue. In reality we discovered that this feature was rarely used because developers didn't link their development schedules to the queue policy.
- We made it possible to easily request reserved nodes on the system for arbitrary amounts of time.

- **Interactive access**. In particular, interactive debugging is important for this type of usage. This type of access to the nodes was enabled by default; when the scheduler allocates a node to a user, that user can login to that node.
- **Property Extraction in job runs**. Development users frequently want information from a variety of sources ranging from kernel counters to network performance data in order to provide insight into execution and performance
- **Permission to stress the system**. Occasionally, large-scale development and testing will tax the system greatly, sometimes beyond the limits of system stability. It turns out that when these sorts of system instabilities occur, developers are usually interested in determining the root cause of the problem and may not consider the instability to be detrimental. In order to support this type of activity, expectations across the user community must be set, i.e. users must be aware that the system will occasionally have instabilities as a result of user code and that this is acceptable (although not actually desirable).

Typically, the intent of the basic development user is to obtain information about performance properties of applications. It should be noted that, in many cases, computational users also carry out basic development when they are developing their computational code and generally have these same requirements.

3.3 System Development

A more demanding type of user of the system is one who carries out "system" development. Projects that require system development capabilities make some kind of modification to the user nodes or to the system itself that require some type of "cleanup" before the system can be used by other users.

System developers typically have the same requirements as basic developers. In addition, they have one or more of these:

- **Root Access**. Some developers, such as those developing device drivers or testing daemons, require privileged access on the user nodes. On a system where root privileges can be assigned to users, the software state on a node can become untrustworthy. Even in the case where users are trusted, honest mistakes can happen, causing a configuration management nightmare. In effect, when root access is granted to users, the node must be considered untrustworthy, and must be rebuilt when the user is complete. One implication of this is that the rebuild process must be robust and efficient. Giving root access to users also has security implications, which are discussed below.
- **Specialized Kernels**. In some cases, developers need a very specific kernel that may be different than the one installed by default on the node. This brings up the same issues as root access does – the node will need to be rebuilt when the user is done.

- **Hardware Management**. Users who are working in this mode are most likely doing work that could crash the node. If this takes place, the user will most likely want access to the hardware of the system in order to debug or restart the node. At the present time on our facility, this would require that the users have access to the management infrastructure of the cluster – a level of access that we are not comfortable making generally available to users. In practice, this situation only comes up rarely, and when it does, it has been possible to have a system administrator participate in the debugging activities. If this becomes a bigger issue in the future, we will need to solve this in a more general way.

Again, the intent of this kind of user is to carry out development and to test properties of their applications, such as stability and performance, rather than to generate any sort of numerical result.

3.4 Extreme Development

The "extreme" developer is one that is developing or packaging up a complete operating system, or is working on cluster-wide system services. Most extreme developers have the same requirements as the previous types of developers, but have one or both of these objectives in addition:

- **User-Defined Node Software**. The user provides an operating system in some form that can be installed on the nodes allocated to them. In order for this to be successful, these images usually need to meet certain requirements: they must be able to use the facility's network, which opens up a number of issues related to node identification. Nodes typically need to set up trust relationships with other nodes in the same project. These issues create a number of technical challenges that are discussed in section 4. Once the user is done with the nodes, they will need to be rebuilt into a standard configuration.
- **Dynamic System Services**. Some projects eventually mature to the point that they can be installed as a part of the cluster fabric for serious testing. Examples of such projects include naming services, mapping services, grid software, file systems, and so on. In every case of this up to the present time, we have had the system managers of the facility get directly involved in the project in order to determine specific goals, testing procedures, and fallback plans. Perhaps the trickiest issue here is that these types of activities tend to destabilize the system infrastructure, once again requiring that the user community have the correct expectations for system reliability.

3.5 Hardware Development

A final type of user in this continuum may be the hardware developer. We have not yet had any hardware developers carry out development or research on our facility, but this is beginning to appear increasingly likely.

Hardware-related projects might include simple testing of new network hardware at scale, augmenting nodes in particular ways (i.e. PIMs or other specialized processors), or trying alternative system management interfaces for nodes.

3.6 Hybrid Models

This categorization of usage models is not precisely discrete. Potentially new approaches to computation and development can be carried out in this type of environment.

For example, it is possible for someone to fine-tune an OS image that is optimized for his or her application, and then to carry out computation using that alternative OS image.

4 Technical Issues

In order to support the technical requirements described above, one must address a variety of issues that are not commonly faced on standard clusters. Some of these can be solved through simple policy changes, for example by adjusting scheduler algorithms and by appropriately setting user expectations. However, many are technical in nature.

In the following sections, we describe the technical issues that we have encountered. In some cases, we feel that we have adequately addressed these challenges in the first phase of this project. In other cases, the problems are beyond the scope of our initial activities.

4.1 Support for Arbitrary OS Images

One of the main requirements of a development cluster, as noted above, is the capability for users to be able to run custom operating systems on the nodes of the cluster. We refer generically to a node's operating system and its configuration as "an image". The degree of image customization varies from user to user and may range from a changed device driver to a completely different OS.

In our environment, we have decided to reinstall the standard node operating system on any node customized by a user, even in the simplest cases. This has worked quite well. We use this same model to rebuild nodes where users have been granted root access (because we have no idea what might have changed).

This node image installation and recovery scheme has three important aspects: an image description mechanism, an OS installation mechanism, and a node recovery system. Each of these will be described in some detail:

4.1.1 Image Description Mechanism

Automated operating system installation has been a common technique in the systems administration community for decades. The most complicated part of this problem has turned out to be the description of the software to be installed on a system and the changes to be made on each individual system. This is also a challenge for clusters.

Fortunately, the systems administration community has developed a variety of configuration management tools and techniques, many of which can be adopted directly for use on a cluster. [3] Despite this activity, image description and change management remain fertile areas of potential research. [4,5]

In essence, images to be installed on nodes must be described somehow. More importantly, users who install their own images must also be able to describe their images, which is an activity that most users would prefer to avoid. The open question in this area is how best to enable users and administrators to easily describe node images, which may include such complications as:

- The node disk geometry
- Other hardware configuration information such as network card parameters
- The base operating system and software packages
- Configuration changes to the base operating system
- Pointers to external services such as naming mechanisms or file systems
- The need to be installed on potentially different nodes over time

We would like a general solution to this problem, but have found that two basic mechanisms have worked for us so far:

- Raw Bit Installation: In this method, an administrator or a user installs an image on a single node. We then take a snapshot of the bits on the node and the disk partition

information. We can then reconstruct an exact copy of that node on nearly any system. We have successfully installed both Linux and Windows operating systems using this method. However, changes in hardware (i.e. devices) or the environment (i.e. servers) limit the effectiveness of this solution.

- Boot Disk Installation: In this method, the user provides a boot disk that will install the proper bits onto disk. The boot disk includes image configuration information that can be customized to each node. That boot disk is distributed onto the cluster management infrastructure, and the individual nodes then boot from that boot disk. (This process is similar to using Kickstart [6], and in fact we have booted nodes using Kickstart in this way.)

We have found that these approaches meet the basic requirements, i.e. they can be used to allow users to install arbitrary images on nodes. However, due to the complex nature of image creation, the cluster administrators usually need to assist with the process. As demand on the system continues to grow, we are concerned that approaches of this type will not scale.

4.1.2 OS Installation Mechanism

The job of the operating system installation mechanism is, as one might expect, to install an image on a node. The installation mechanism should be able to install an arbitrary image (i.e. any image built in the ways described above).

The OS install mechanism should not rely on any hooks or infrastructure on the node itself; it should work if the node has no software, has a supported operating system, or is running mysterious user code.

Fortunately, the industry has largely solved this problem with standard network booting protocols. DHCP [7] and PXE [8] are commonly used to remotely install operating systems on computers over a network. PXE, unfortunately, only works on a subset of Ethernet cards, however these are increasingly common.

In addition to these standard protocols, our solution incorporates a database that maps nodes to desired images, making boot decisions based on the state of the system. In some cases, we will install the appropriate image directly. In others, where this is not possible, we will install a known reliable image that will format the local hard drive appropriately and then install the target image.

4.1.3 Node Recovery System

Once user jobs have run, nodes may have been left in an indeterminate state. The job of the node recovery system is to prepare those nodes for image installation. Because the node state is unknown at this point, the infrastructure for node recovery must work independently of host operating system support.

As our boot management system can take over during a reboot, the simple solution is to force a reboot of the node. We have implemented this with the use of network accessible power controllers.

This solution works fairly well for us, however:

- If a user were to reset the BIOS so that a network boot was not forced, the node would not recover correctly. If this were done across the cluster, we would be forced to manually reset the BIOS on every node, which would be a real disaster.
- We're beginning to worry about repeatedly interrupting the power to the nodes. This doesn't seem to be the best way to treat hardware. We would prefer to have hardware-

level reboot and power cycle control, such as exists as a management interface on some systems.

4.2 Complex Infrastructure Requirements

In many cases, developers who are working on the system are able to work on any available set of nodes without substantial adaptation of their code. In general this is true for developers in the Basic Development and Systems Development categories. However, some developers discover that they have to think carefully about adapting their project to a dynamic cluster environment. This is frequently the case for developers who make substantial changes to the node environment or are working with cluster-wide system services, i.e. Extreme Developers, and also occasionally an issue for others.

4.2.1 Entire Cluster Simulation

One of the popular uses of our facility is to test cluster operating systems and cluster management software. Configuring the test environment for this kind of use takes special care. Typically, these developers need to simulate an entire cluster, including management, login, storage and compute functionality, all within the user nodes allocated to them.

This is more difficult than normal for two reasons:

- The users need to map their services onto a set of basically identical systems. The concept of generically mapping services onto nodes (rather than onto some hardwired test cluster) is usually foreign.
- The users often need some specific hardware configuration. This may require more disk space than is available (i.e. to simulate storage nodes), different network cards than the nodes are configured with, or different hardware arrangements (i.e. to access serial consoles or hardware performance data) than is possible.

We do not have a general solution to these problems at this time. We are leaning towards having a pool of "advanced capability" nodes, i.e. with extra memory, disk, and room for peripherals in order to address the second issue.

4.2.2 External Infrastructure

Some of the software under development on the cluster relies on persistent, external infrastructure. Those persistent services could conceivably be installed dynamically as parts of that user's "job", but in practice are much easier to install once on some dedicated system. These often help bootstrap the user's code.

We have two distinct examples of this:

- When installing Windows2000 as a cluster OS environment, it is useful to have at least one Windows Domain Controller accessible on the cluster network, providing Windows-specific information such as user accounts, naming, and so on.
- In order to facilitate some of the Red Hat-based distributions that have been installed on the cluster, we allocated a dedicated computer to act as the Kickstart boot manager.

The ability to provide external services on a persistent system has simplified these two experiences substantially.

4.2.3 Production Integration

In some cases, the software under development is approaching production-ready quality. The next step for these projects is to try the software with real users in production mode. The goal at

this point is to flush out remaining bugs and understand performance issues under a real workload. The logical next step is to deploy this software as part of the real cluster infrastructure.

On our system, this has taken place with file systems, process managers, and messaging libraries. The system administrators of the facility installed these on the cluster itself, and these software versions became the default versions available for users. At this point, the users of the cluster a whole (both computational and development) became the set of users testing these software packages.

This process raises three issues. First, it's not necessarily clear that this is a good idea to do with all software, and the selection criteria are a bit unclear. Second, doing this requires a great deal of interaction between the project developers and the cluster administrators; both parties need to buy in to the plan. Finally and perhaps most importantly, the resulting instabilities can cause problems for all users on the cluster. As we noted previously, it helps to warn users in advance that these kinds of failures are expected.

4.3 Supporting Data Gathering in Jobs

We have found that development users want information from a variety of sources to provide insight into application execution and performance. This data includes:

- Kernel counters
- Hardware counters, such as cache hits, and number of instructions executed
- Environmental data such as temperature, clock speed, etc
- Network performance data
- Application profiling data

This data is accessible in locations both on and off of a user node. For this reason, the application cannot necessarily collect at of this data directly. At this point, we provide the data on an ad-hoc basis. A more general solution would be very helpful.

4.4 Persistent Node Identification

A possible characteristic of a development facility is that hardware configurations change fairly regularly, either as a part of testing hardware or when replacing nodes. In our case, we've learned that we shuffle hardware far more than we had originally anticipated.

On a system in which individual nodes are commonly moved or replaced, the issue of node identification becomes both important and difficult. In this sense, "node identification" is the ability to permanently map a node's physical location in the cluster to a hostname. Knowing the exact physical and network location of a node (and its associated hostname) is important for many reasons, including:

- The ability to locate a node when hardware problems occur.
- The desire to track performance and reliability trends based on a persistent name associated with the hardware.
- The fact that both developers and administrators occasionally need to understand the topology of the system, for example, when analyzing network performance characteristics, or when renumbering subsets of a cluster for security reasons.
- On a development cluster, it is highly desirable to know a node's identity before it boots in order to deliver the correct OS to that node.

Knowing specifically which hardware has which network address sounds like a very simple problem, but it is difficult when hardware is regularly moved or changed. Most name assignment

schemes are based on the theory that one only assigns names once, or only assigns names dynamically. Neither is true in this situation.

The usual method for assigning names is to gather the MAC addresses of each node, associate those addresses with known hostnames, and then to assign them to nodes dynamically via DHCP or statically via some host configuration system. On any cluster that is only installed once, MAC address gathering can be done by hand or via a controlled sequenced boot of each node, one by one.

In a cluster with dynamic hardware configurations, a more reliable solution is to have the cluster infrastructure detect that hardware has changed.

Our initial implementation of this on Chiba City used the serial console infrastructure to detect MAC addresses, register then in the cluster's DHCP services, and modify the node installations as necessary. This was an option because the serial console port could be used to precisely identify physical location of the node. Because the serial console hooks only work on our own node image and not on arbitrary node images, we are now working on a more general solution that uses network switch sensing to detect MAC addresses.

4.5 Node-Proof System Software

On a system on which the user nodes may be running any kind of code, the system infrastructure must be able to function optimally without any dependency on those nodes. We have found in particular that some scheduling and job launching software tends to rely on status information from daemons on the user nodes, and will hang or time-out when those nodes are unresponsive.

4.6 Smarter Node BIOS

Standard commodity PC BIOS systems are fairly limited. This has caused problems for us in a number of ways. We would find the following features extremely helpful:

- Accessibility to the BIOS over the console port.
- Operating system access to the BIOS, i.e. being able to set BIOS features in the same way that is possible on Solaris systems and others.
- Enhanced monitoring capabilities.
- A consistent approach to locking BIOS settings with a password.

Some of these BIOS issues are being addressed by the LinuxBIOS [9] project (which also has other desirable features). Unfortunately, in our case, the hardware that we are using cannot take full advantage of LinuxBIOS, thus we are also hoping for a wider acceptance of LinuxBIOS by the vendor community in order to be able to use it in the future.

4.7 Security

Allowing users to have root access introduces a number of complicating issues.

The most common problem that we have faced is that someone with root may have done something (intentionally or not) to modify the state of the image. As noted in section 4.1, we solve this by rebuilding a node from scratch after a user with root privileges has finished with it.

Users with root access also introduce a number of security issues, particularly if there is a possibility that the user may be malicious:

- Nodes user administrative control of users should not be trusted by system management infrastructure. So, for example, a cluster's NFS file systems should not be exported to such nodes.

- Some standard trust-related configurations, such as cluster-internal ".rhosts" or .ssh key files, are no longer an option. Nodes under the control of one user should not trust nodes under control of another user.
- Users might use their nodes to launch attacks against other networked systems outside of the cluster. (A cluster would make an interesting testbed for denial-of-service attacks.)
- Users might introduce network-based attacks such as IP spoofing or network sniffing in order to attack some other job currently running.
- Users may configure system services (intentionally or otherwise) such that their nodes are susceptible to attacks from outside.

We have not yet addressed any of these issues yet because our user community we are quite familiar with that portion of our user community to whom we have granted root access, due largely to their need for aid with image configuration issues. Thus, we have been able to keep an eye out for such activities. However, this is becoming an increasingly important issue as our user community grows.

These problems can be addressed in the following ways:

- Nodes upon which a user has root access should be designated as "untrusted".
- Untrusted nodes should be easily discernable from trusted nodes in order to allow the management infrastructure to differentiate between the two. In particular, they should have different IP addresses and hostnames than when they are trusted.
- In order to prevent network-based attacks within the cluster, untrusted nodes should be on different networks from trusted nodes, and also from other groups of untrusted nodes. This segmentation requires the use of routers between all distinct security zones.
- Network filters and detectors should be installed to detect attacks originating in the cluster and targeting the cluster. Segregating trusted and untrusted hosts by network can help with these filters.
- Clearly, any detected security problems should immediately result in the disabling of the associated user's access to the cluster.

These solutions require networking gear that can operate at both layer 2 and layer 3, perform IP filtering on the fly, and can be dynamically reconfigured in associate with node allocation on the cluster.

5 Conclusion

In this paper, we have described the technical challenges that we have encountered in the first phase of a project to build and operate a cluster in support of development and scalability research. Facilities to support large-scale development and research are critical to the future rapid growth of HPC systems and the HPC scientific community.

A number of the challenges in supporting this facility were solved with a robust node management infrastructure and with flexible system policies, but the most difficult problems, ranging from image description through data gathering remain. As we add more users to the system and support increased functionality, it will become imperative that we solve these in a general way.

6 References

[1] R. Evard, "Chiba City: A Case Study of a Large Linux Cluster", in *Beowulf Cluster Computing with Linux,* by Thomas Sterling. MIT Press, 2001.

[2] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. “BEOWULF: a Parallel Workstation for Scientific Computation”, in *Proceedings of the 24th International Conference on Parallel Processing*, 1995.

[3] M. Burgess, “Cfengine, a Site Configuration Engine”, in *USENIX Computing Systems* Volume 8, 1995.

[4] Putchong Uthayopas, Sugree Phatanapherom, Thara Angskun, Somsak Sriprayoonsakul, "SCE: A Fully Integrated Software Tool for Beowulf Cluster System", in *Proceedings of Linux Clusters: the HPC Revolution*, National Center for Supercomputing Applications(NCSA), University of Illinois, Urbana, IL, June 25-27, 2001.

[5] Philip M. Papadopoulos, Mason J. Katz, and Greg Bruno, “NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters”, in *Proceedings of Cluster 2001*, October 2001.

[6] Kickstart: http://www.redhat.com/docs/manuals/linux/RHL-7.3-Manual/custom-guide/

[7] DHCP: ftp://ftp.isi.edu/in-notes/rfc2131.txt

[8] PXE: ftp://download.intel.com//labs/manage/wfm/download/pxespec.pdf

[9] R. Minnich, J. Hendricks, and D. Webster, “The Linux BIOS”, in *The Fourth Annual Linux Showcase and Conference*, October 2000.

License Statement (not meant to be published)