Easy and Reliable Cluster Management:

The Self-management Experience of Fire Phoenix*

Zhang Zhi-Hong, Meng Dan, Zhan Jian-Feng, Wang Lei, Wu Lin-ping and Huang Wei

Institute of Computing Technology
Chinese Academy of Sciences
P.O.Box 2704 Beijing, China 100080
{zzh,dm,jfzh,wl,wlp,hw}@ncic.ac.cn

Abstract

High-Performance clusters are rapidly becoming an important computing platform for both scientific and business applications. To fulfill the new demands and challenges, cluster system software is inevitably complex. Even for experienced administrators, the management of a cluster system is an exhausting job. This paper introduces Fire Phoenix, a scalable and self-managing cluster system software that supports both scientific and commercial applications. With the self-configuring and self-healing features, much of the machine configuration and error recovery can be done automatically. Our design has been proven effective in the operations of the Dawning 4000A supercomputer, which is the biggest cluster system in China.

1. Introduction

Within the last decade, clustering has become one of the mainstream technologies with its low cost and high scalability. Not limited in traditional scientific computing applications, clusters are more and more widely adopted in business domains including databases, large web sites, and digital libraries. Compared with scientific computing, the business domains have different requirements, such as scalability, reliability and support for heterogeneous platforms. Facing these challenges, cluster system software should have a flexible component framework, high availability service and dynamic configuration ability to meet the growing and varying demands of business applications; at the same time, the cluster system software should hide the complexity and make cluster management easy and reliable so as to avoid operation errors and reduce the TCO.

Traditional cluster management software [1-3] targeted on scientific computing lacks the support for business applications. Facing this challenge, we developed a new cluster management software named Fire Phoenix, which supports both scientific and business computing on heterogeneous platforms. To make system management easy and reliable, we borrowed autonomic computing principle and enabled system self-management. In this paper, we describe how we design and implement the self-management mechanism and evaluate it from several aspects. The research contributions of this paper can be concluded as:

1. Providing an agent-based solution to self-configuration of heterogeneous cluster resources,
2. Proposing a role-based self-deployment mechanism for cluster service components,
3. Proposing a booting protocol, which turns the unstable initial state of cluster system software into a self-healing one,
4. Providing a group service based self-healing solution for high availability of cluster so that every system element can be recovered from faults and no single point of failure exists.

The remainder of the paper is organized as follows. Section 2 outlines the related work. Section 3 gives the overview of Fire Phoenix. Sections 4 and 5 describe and

* - This work is supported by the National '863' High-Tech Program of China (No. 2004AA616010) and the 15th key project of China (No. 2004BA811B09-1)

1-4244-0054-6/06/$20.00

analyze the self-configuring and self-healing mechanism of Fire Phoenix respectively. Section 6 draws a conclusion.

2. Related Work

The configuration and deployment of a large-scale cluster is an exhausting work. Many earlier cluster management systems attempt to tackle this problem. NPACI Rocks [1], an open source Linux-based cluster solution package, can help significantly reduce the complexity of building Beowulf HPC clusters. It makes use of the Redhat's Kickstart and RPM to manage the distribution of node file system and provides component-based configuration method to realize module reuse. OSCAR [2] (Open Source Cluster Application Resources) is a snapshot of the best-known practices for building, programming, and using clusters. It consists of a fully integrated and easy to install software bundle designed for high performance cluster computing. Warewulf [3] is a light-weighted cluster toolkit that facilitates the process of installing a cluster and long-term administration. These pioneer researches forward the target of easy and reliable management of cluster system.

However, the above-mentioned studies cannot fulfill the demands of high availability and varying configuration. IBM's RSCT [4] (Reliable Scalable Cluster Technology) is the infrastructure used by a variety of IBM products to provide clusters with improved system availability, scalability, and ease of use. Now it can provide a comprehensive cluster environment for AIX and Linux.

With the increasing scale and complexity of clusters, cluster system software becomes more complex. Autonomic computing presents a new way to tackle this issue. Inspired by the functioning of a human nervous system, automatic computing is to build and design computer systems that function like it. The object of automatic computing is self-management, i.e. self-configuration, self-optimizing, self-protection and self-healing. In this way, a computing system can hide the complexity from users and make them concentrate on high-level management objectives instead of low-level management operations. The papers [5, 6]conceives the picture of autonomic computing and outlines the architecture upon that an autonomic system might be built. [7]Believes that the way in conquering autonomic systems is the integration of three existing research communities: the multi-agent systems community allows natural modeling of the system and explicitly considers autonomous behavior and distributed interaction, dynamical systems theory allows analysis of the dynamics of these models and the decentralized control community can use insights gathered from analysis to create decentralized control mechanisms to control the dynamics of autonomic systems. A few approaches show the promise of using a comprehensive architecture to address the modeling of autonomic computing, but have not realized the promise.

3. Overview of Fire Phoenix Cluster Operating System

Fire Phoenix Cluster Operating System [8] (in short, Phoenix) is a cluster system software, which is designed and developed from the perspective of an operating system, and aims to provide a scalable and highly available distributed heterogeneous platform to support both scientific and business applications. Like a UNIX operating system, Phoenix has a 3-layered architecture. The top layer is the user environment, through which users utilize cluster resources to fulfill their requirements. The second layer is Cluster Operating System Kernel (in short, Phoenix Kernel), which defines the minimum set of core functions with scalability and fault-tolerance support. The lowest layer is the heterogeneous resource layer, which shields heterogeneous hardware architectures, host operating systems and communication protocols with heterogeneous middleware. Figure 1 shows the main components in Phoenix Kernel layer. The green blocks indicate modules responsible for self-configuring and blue blocks indicate modules responsible for self-healing.

4. Self-configuring mechanism

4.1 Scenario and Motivation of Self-configuring

Phoenix supports several heterogeneous platforms, including Linux, Windows, Solaris and Aix. When we talk about the deployment of Phoenix, we assume that a host operating system has already been installed.

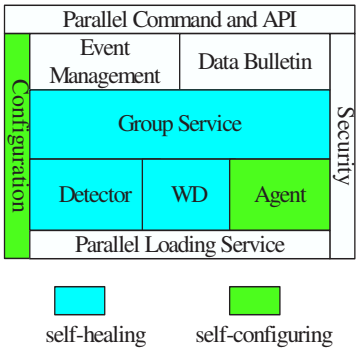

Figure 1. Layered architecture of Phoenix Kernel

To deploy Phoenix from scratch, the following steps are needed:

1. Configuration: gathering the cluster hardware information, such as number of nodes，physical resources and network configuration, and planning how to deploy different service components on each node,
2. Deployment: installing service programs on every node according to the configuration,
3. Booting: bringing up service programs according to the dependences between services,
4. Maintenance: ensuring the correctness and consistency of the running environment on every node.

Originally we designed and implemented a series of tools to do these work:

1. A GUI tool named Configuration Center is used to edit the configuration of the cluster.
2. Several shell scripts based on RSH and RCP commands are used to install the service components.
3. A GUI tool named Control Center is used to bring up service components in order.
4. If any problem occurs, the manager is responsible for maintenance.

We call this solution as the GUI+Scripts mode. This solution has several shortcomings: Firstly, configuration, deployment, booting and maintenance are separated and the manager must do a lot to glue them into an integrated process. Secondly, all the operations based on RSH or RCP commands are sent from one management console and highly centralized management leads to low efficiency. Thirdly, remote shell commands fail to interoperate in heterogeneous environments.

To overcome these shortcomings, we present an agent-based solution, which works in a distributed and cooperative way to provide full support to heterogeneous platforms. This solution can greatly ease the management by means of self-configuration.

An agent is a software entity that is capable of taking autonomous actions to meet its design objectives in a situated environment. We believe an agent-based solution can meet the demands of cluster construction. Firstly, agents, situated in nodes with different hardware architectures and host operating systems, can detect resources and accommodate to circumstances, and hence support the cluster construction in heterogeneous platforms; secondly, agents, taking actions autonomously, make cluster construction easier.

4.2 Architecture

As Figure 2 shows, Phoenix s self-configuring mechanism includes 4 elements: Decision-making Module, Configuration Service, Deployment Service, and Node Agent.

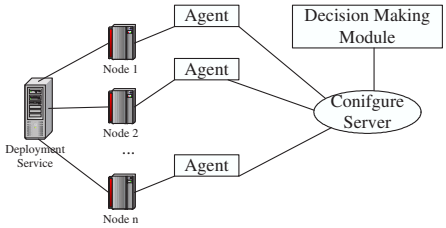

Figure 2. Architecture of phoenix self-configuring mechanism

4.2.1 Decision-making Module

This module decides how to configure cluster resources and services according to the cluster information stored at Configuration Service. Decision-making Module may be designed in two ways. In the first way, a human manager makes decisions and this module is actually a convenient GUI tool; and in the second way, the module will be a daemon that can make decisions autonomously. We compromise these two ways: a GUI tool displaying the system information helps the manager make decisions, and a daemon executes the decisions. In this way the manager may absolutely control the system or make some rules to direct the daemon to work automatically when the conditions are met.

4.2.2 Configuration Service

The Configuration Service provides calling interfaces for other applications to access the cluster configuration information. In most implementations of existing cluster management systems, managers need to manually input the configuration information. The manual work is annoying and error-prone especially when the scale of cluster enlarges and when the configuration changes dynamically. In Phoenix, the Node Agent automatically detects the local machine configuration and reports it to the Configuration Service. Then the Configuration Service can edit this information into a uniform format for access.

4.2.3 Deployment Service

The Deployment Service provides program packages for access. In our implementation, the Deployment Service is an FTP server and the Node Agent can access program packages through ftp protocol. To support heterogeneous hardware architectures and host operating systems, the Deployment Service stores packages according to a hierarchical directory structure as Figure 3 shows.

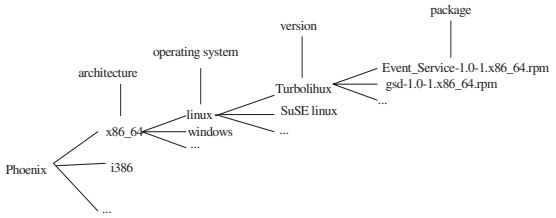

Figure 3. Hierarchical directory structure in Deployment Server

Along with a package, Deployment Service also keeps a text file, named recipe, which provides instructions on how to install and bring up a service program properly.

Figure 4 shows the recipe format of one service component named Event_Service on Linux system.

4.2.4 Node Agent

Node Agent is the key module of this architecture. It can cooperate with other modules and take the actions of configuring, installing, booting and maintaining.

In traditional computing clusters, the role of a node is simply classified as either a computing (or slave) node or a management (or master) node, where a large number of computing nodes carry out high density computing task and a few management nodes manage the computing node. For example, Chiba[9], a 312-node cluster，is divided into several parts called towns. Every town is composed of 8-32 computing nodes and one master node called mayor. The mayor carries out all services to manage the computing nodes in the town. And a master node called president manages all mayors in higher hierarchy. The main lesson one learns from Chiba is that statically linking managed nodes to a specific master node leads to poor server load distribution and single point of failure.

In Phoenix, the role of a node is defined as a combination of services and we classify the role of a node according to the services running on this node. For example, if node N is configured to run service S, we can say node N has a role including S. In this way services are not statically linked to specific nodes, but are able to be dynamically configured on any nodes and to serve any requesting nodes. The role of a node will change when the services configured on it change. When a service is ready to be configured and deployed in Phoenix, it will be assigned a unique ID less than 64. So we can define the role of node in the following way:

The Role of Node is represented by a 64-bit integer, in which every bit represents a service ready to run on Phoenix. If one service is configured on the node, the corresponding bit defined by its unique ID will set 1, else 0.

```
id 4
name=Event_Service
version=1.0
path=/cluster/phoenix/Event_Service/
boot ./rs start.sh
shut=killall Event_Service
package=rpm
install=
uninstall=

      Event_Service.recipe
```

Figure 4. Recipe example of Linux version of Event Service

In this way the assignment, comparison and update operation on a role can be carried on flexibly and efficiently by binary AND, OR and XOR operation.

4.3 Self-configuring Process

The self-configuring process is triggered by Node Agent s automatic actions, which are guided by Decision-making Module. The actions of Node Agent are composed of the four steps: configuration, installment, booting and maintenance. These actions may alter the state of node, hence the whole cluster system. The Figure 5 shows the state-transition of a Node Agent.

Configuration: In this step, the function of Node Agent is somewhat like that of the BIOS system in PC. When a Node Agent starts, it will make a self-test to detect the resources in the local environment, including hardware, host operating system and network configuration. Then it will report the information to the Configuration Service. Configuration Service edits the information and notifies the Decision-making Module. The Decision-making Module assigns a role to this Node Agent according to the resources of the node.

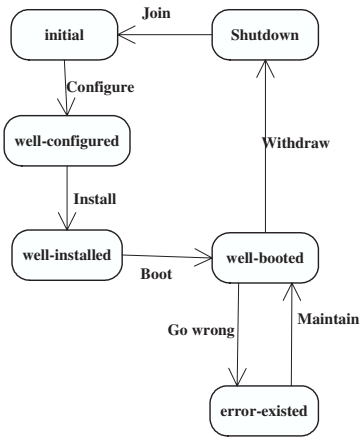

Figure 5. State diagram of a Node Agent's actions

Installment: After a Node Agent joins, it will accept a role from the Decision-making Module, representing which services should be running in this node. According to its role, the Node Agent will download service recipes and program packages from the Deployment Service. Then it will install programs and make proper configuration for the programs in the local node instructed by the service recipes.

Booting: Like any other operating system, Phoenix needs several steps to bring up all Phoenix service components. From this perspective, the Node Agent in Phoenix is somewhat like the Init process in Linux. Only after a successful bring-up can the Phoenix kernel transit from unstable initial state to self-healing state; hence a reliable booting protocol is necessary.

Main kernel services, such as Group Service Daemon (in short, GSD), Event Service and Data Bulletin Service, need to work as a group. The construction of a group is a process of collective startup, which means the booting of service group should keep atomicity and consistency. Hence first of all, we design a group-booting protocol to guarantee reliable group startup as follows:

Step 1: Decision-making Module nominates one Node Agent as a commander to coordinate the booting; then the commander queries the Configuration Service as to which nodes are configured as group members of this group;

Step 2: The commander issues booting commands to the Node Agent one by one according to the rank of group members. The Node Agents which receive the commands are called executors. After sending all commands, the commander sets a timeout to wait for replies from the executors;

Step 3: An executor brings up its group member through application programming interfaces provided by the local operating system. Then it sets a timeout and waits for the message replied from the specified group member;

Step 4: Once the group members bring up, they will communicate with each other to create a group. After the group establishes successfully, every member will send a success message to the local executor.

Step 5: If any one of the executors receives the reply in time, it will send a success message to the commander; and if a timeout message arrives, it does not send back any message and just returns to wait for a new command;

Step 6: If the commander receives a reply from any one of the executors, the group boots well; if no message comes before timeout, the booting fails.

Taking into the account of dependencies among different service components, the booting algorithm of Phoenix Kernel [8] can be described as follows:

Step 1: The administrator issues the booting command through the GUI of the Decision-making Module;

Step 2: The Decision-making Module nominates one Node Agent as the commander responsible for this booting process; then it sets a timeout to wait for the commander s reply;

Step 3: The commander boots GSD using the group booting protocol. If the booting fails, it will send all executors a shutdown command to cancel the booting and send a message to the Decision-making Module about booting failure with GSD;

Step 4: The commander boots Event Service and Data Bulletin Service using the group booting protocol in the same way as GSD was booted;

Step 5: The commander notifies all Node Agents to bring up Watch Daemon (in short, WD) and Detector in its local nodes;

Step 6: The commander replies a success message to the Decision-making Module;

Step 7: If the Decision-making Module receives the success message from the commander before the timeout, the Phoenix system boots successfully; and if the failure message or the timeout comes, it will show the failure message to the administrator.

When the system boots successfully, all groups were created successfully. The Phoenix system turns into a self-healing state, which will be described in detail in next section.

Maintenance: Other than the three actions discussed above, the Node Agent is responsible for the maintenance of services running in the environment. Every time before a booting operation starts, an agent will examine if the necessary environment variables are configured well and if programs are installed properly. If any fault is discovered, the agent will try to recover it by reinstalling the program.

4.5 Evaluation

We evaluate the self-configuring solution from two aspects: one is the ease of management and the other is the efficiency compared with the original GUI+script implementation.

4.5.1 Evaluation of ease of management

The management can be measured by 3 main operations: deploying Phoenix from scratch, dynamically adding a node and dynamically adding a service.

To deploy Phoenix from scratch, a little manual work is needed to install self-configuring modules, including

Configuration Service, Deployment Service, Decision-making Module and Node Agent. Though installing Node Agent on every node sounds annoying, we get it done once and forever. Then the remaining thing is to assign roles to nodes, which is easy to understand and can be done only by several clicks in GUI of Decision-making Module.

Dynamically adding a new node into the cluster is very easy in self-configuring mode, which includes two steps: installing and running a Node Agent, then assigning a role to it.

Adding a new service needs a service recipe and the installment package of the service. The Decision-making Module assigns a unique ID to this service and setups the service recipe and package into the Deployment Server. Then the Decision-making Module creates a new role including this service and assigns this role to the nodes. This frame is not specific to the Phoenix kernel service, but is also suitable for any service in the top user environment layer if only appropriate service recipes and packages are provided.

In the three operations, most work can be done automatically in spite of the necessary initial installment and role assignment. From the experiences of users, they usually can understand and master this way of management in hours.

4.5.2 Evaluation of efficiency of Deployment

We tested the performance of deploying Phoenix from scratch on two cluster platforms. The first one is Dawning 4000A, a 532-node homogeneous cluster running Turbo Linux 8.0. Another one is an 8-node heterogeneous cluster including 3 kind of hardware architecture: 2-way NUMA with 64-bit AMD Opteron CPU, 4-way SMP with 32-bit Intel Xeon CPU and PC with Pentium 4 CPU. The heterogeneous cluster is hosted with 3 kinds of operating systems: Turbo Linux 8.0, Redhat Linux 9.0 and Windows 2000 Professional edition.

In this test, the cluster software packages to be deployed include all Phoenix kernel services, a job management system and a Business Application Runtime Environment. We compare the deployment time of self-configuring mode with original GUI+scripts mode in Table 1 and Table 2.

	Configuration (minute)	Installment (minute)	Booting (minute)
GUI+Script	30	60	5
Self-config	5	10	2

Table 1. Deployment time in Dawning 4000A

	Configuration (minute)	Installment (minute)	Booting (minute)
GUI+Script	10	10	1
Self-config	3	2	0.5

Table2. Deployment time in an 8-node heterogeneous cluster

From the comparisons, we make sure that the agent-based solution can save a lot of time in deployment with autonomous actions and better parallelism.

5. Self-healing Mechanism

The self-healing module works in face of omission failures of nodes, communication channels and processes. The group service [10] is a fundamental component that makes Phoenix a scalable and highly available cluster system software.

5.1 Group Management Framework

In Phoenix system, the whole cluster is divided into several cluster partitions, each of which consists of at least two nodes with the role of GSD (Group Service Daemon): one node is responsible for management of the partition, and another is its backup in case of node failure. Several GSDs form a meta-group, which is created and managed by the membership protocol. After having been configured and booted, the service groups can build up and enable self-healing. We discuss the self-healing mechanism from three aspects: self-healing at the group level, self-healing at the node level and self-healing of cluster services with important data.

5.2 Self-healing at the Group Level

The self-healing mechanism of GSD is the basis of that of Phoenix. As the Figure 6 shows, the GSD-meta group takes a ring structure. Every group member sends periodic heartbeat to its neighbor in the ring.

Once one member does not receive the heartbeat from its neighbor in time, it will report the suspected failure about its neighbor to the leader. The leader will decide whether the suspected node or process has failed. If it is a process failure, the leader can simply recover it by rebooting the application. If it is a node failure, the leader will ask the member who detects the failure to take over the job of the failed one. Then it will bring up a new GSD on the backup node in that partition. Once the backup one is up and joins the group, it will take back its job again.

The one who monitors the leader is called the prince. When the leader fails, the prince will become the new

leader and the one next to it in the ring becomes a new prince.

The GSD also provides APIs for creating application-groups. In the kernel layer, Data Bulletin Service and Event Service call these APIs to create a service group and register policies on how to deal with failures.

As Figure 7 shows, GSD takes charge of monitoring the aliveness of the Event Service. If one member of the Event Service group fails, the GSD on the same host will notify all members of the GSD group and then restart the failed service. If the node on which the Event Service daemon is running fails, the GSD member next to it in the ring structure will select a new node for migrating the GSD and then recovering the Event Service.

5.3 Self-healing at the Node Level

Watch Daemon (in short, WD) is running on every node in Phoenix system and provides high availability for all local applications in the same node. WD sends periodic heartbeat to GSD and GSD takes charge of all WDs in its partition. Once there is some failure in a WD, GSD can reboot it. Figure 8 shows the relationship between GSD and WD.

WD provides a self-healing interface for local applications. An application can call the interface of WD to register its failure handler. After registration, WD will hire Detector (Detector is a daemon responsible for detecting information about resources and applications of a node) to monitor the application state. Once Detector discovers an application failure, WD will reboot the application with the failure handler that the application had registered.

5.4 Self-healing of Configuration Service and Deployment Service

Configuration and Deployment services are important for Phoenix. Their failures cause a single point of failure to Phoenix system and make dynamic configuration impossible. So we should guarantee both the high avalibility of application process and important data.

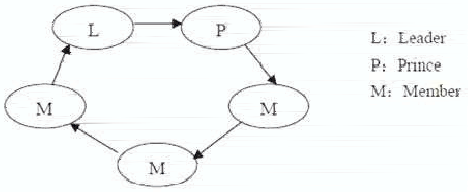

Figure 6. Ring of Group Services

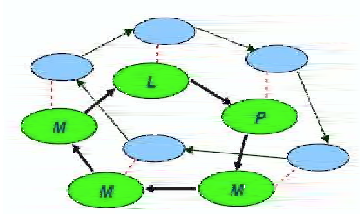

Figure 7. Event Service Group based on GSD

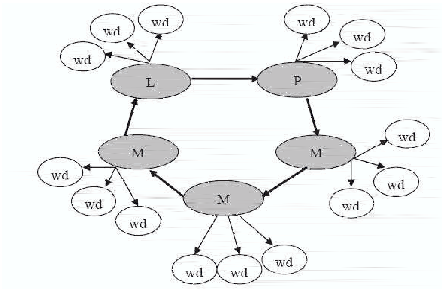

Figure 8. Relationship between GSD and WD

Figure 9 shows the solution for this problem, where a Raid disk [11] is needed to keep the data highly available. Two nodes, one master and one backup, connect to Raid through SCSI interfaces. In the Figure 9, CS stands for Configuration Service and DS stands for Deployment Service.

These two nodes should be configured into one partition, so that one GSD monitors both of them. At any time only the master process works. In case it fails, the local WD is responsible for rebooting it; if the master node is down, GSD will detect the node failure in time and notify the WD to bring up the backup service on the backup node. Because Configuration Service and Deployment Service have no state and all data comes from the reliable Raid disk, the new service can continue the work when the failed one is checked and repaired.

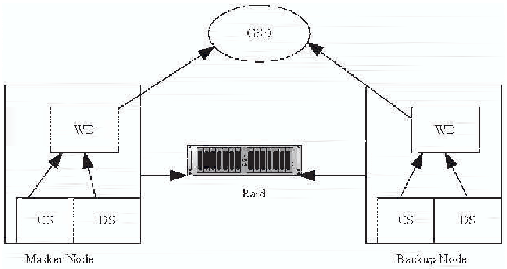

Figure 9. Duplex machines highly available solution for Configuration and Deployment Service

5.5 Self-healing Evaluation

The main performance criteria for evaluating fault-tolerant system software are failure detection overhead and failure recovery overhead. In this study, we have conducted experiments to measure these metrics of the proposed framework. The testbed is composed of 136 nodes from 8 partitions, with 18 nodes per partition, on the platform of Dawning 4000A. The interval for sending heartbeat is configured as 30 seconds for testing.

By the means of fault injection, we get the information about nodes, groups and applications failure detection time and recovery time respectively. From the data, we can conclude that the sum of failure detection time, diagnosis time and recovery time is almost equal to the interval of sending heartbeat, while the interval for sending heartbeat can be dynamically configured. It proves that Phoenix kernel has good self-healing ability. From the analysis above we can see that Phoenix provides a self-healing mechanism from bottom up and makes sure that any omission failures about process, network and node can be found and recovered in time.

6. Conclusion

This paper introduces Fire Phoenix, a cluster management system software supporting both scientific and business applications on heterogeneous platforms. We discuss the challenges in the development and management of Phoenix and provide our self-configuring and self-healing solutions to meet these challenges. The four contributions (agent-based configuring mechanism, role-based deployment, group booting protocol and group service based self-healing mechanism) are presented and discussed in detail. Practical experiments show that our work can deliver easy-to-use and reliable cluster system management tools to the users.

In this system, some configuration work, such as designing role and making deployment plan, is not yet mature. Our future effort will involve injecting more intelligent behaviors into the Decision-making Module to improve the self-configuration ability. In the self-healing aspect, Phoenix can only recover a stateless process. Our future research will focus on the self-healing of stateful applications.

References

[1]Philip M. Papadopoulos, M.J.K., and Greg Bruno. NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters. *In Proceedings of 2001 IEEE International Conference on Cluster Computing*. Newport. 2001

[2]Soctt., S.L. OSCAR and the Beowulf arms race for the "Cluster Standard". *In Proceedings of 2001 IEEE International Conference on Cluster Computing*. Newport,CA. 2001

[3]Labratory, L.B.N., The Warewulf Cluster Toolkit, http://warewulf.lbl.gov/pmwiki/, April 2005.

[4]IBM Reliable Scalable Cluster Technology http://publib.boulder.ibm.com/infocenter/clresctr/index.jsp?topic=/com.ibm.cluster.rsct.doc/rsctbooks.html

[5] J.O. Kephart and D M Chess, The vision of autonomic computing. *IEEE Computer*, 42: 41-55, 2003.

[6]T.A.Corbi, A.G.G.a., The dawning of the autonomic computing. *IBM System Journal*, 42: 5-18, 2003.

[7] Tom De Wolf and Tom Holvoet, Towards Autonomic Computing: Agent-Based Modeling, Dynamical Systems Analysis, and Decentralized Control. *In Proceedings of the First International Workshop on Autonomic Computing Principles and Architectures*.2003

[8]MENG Dan, Zhan Jianfeng, Wang Lei, TU Bibo and ZHANG Zhihong, Fully integrated Cluster operating system: Phoenix. *Journal of computer Research and Development,* 42(6):979-986， 2005.

[9]John-Paul Navarro, R.E., Dan Nurmi, Narayan Desai. Scalable Cluster Administration Chiba City I Approach and Lessons Learned. *In Proceedings of 2002 IEEE International Conference on Cluster Computing*.2002

[10]Huang Wei, Zhan Jianfeng, and Fan Jianping, DCFT-Kernel: A Fault-Tolerant Cluster Middleware Based on Group Service. *Journal of computer Research and Development*, 42(6): 993-999, 2005.

[11] P.M.Chen, E.K.lee, G.A.Gibson, R.H.Katz, and D.A.Patterson, RAID: High-perfornamce, Reliable Secondary Storage, *Acm Computing Surveys*, Vol.26, No.2, 145-185, June 1994.