Li Y, Chen FH, Sun X *et al.* Self-adaptive resource management for large-scale shared clusters. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 25(5): 945–957 Sept. 2010. DOI 10.1007/s11390-010-1075-6

Self-Adaptive Resource Management for Large-Scale Shared Clusters

Yan Li (李 研), Feng-Hong Chen (陈峰宏), Xi Sun (孙 熙), Ming-Hui Zhou (周明辉)*, *Member*, *CCF*
Wen-Pin Jiao (焦文品), *Senior Member*, *CCF*, Dong-Gang Cao (曹东刚)
and Hong Mei (梅 宏), *Senior Member*, *CCF*

Key Laboratory of High Confidence Software Technologies, Ministry of Education, Institute of Software
School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China

E-mail: {liyan05, chenfh10, sunxi, zhmh, jwp, caodg}@sei.pku.edu.cn; meih@pku.edu.cn

Received April 1, 2009; revised February 1, 2010.

Abstract In a shared cluster, each application runs on a subset of nodes and these subsets can overlap with one another. Resource management in such a cluster should adaptively change the application placement and workload assignment to satisfy the dynamic applications workloads and optimize the resource usage. This becomes a challenging problem with the cluster scale and application amount growing large. This paper proposes a novel self-adaptive resource management approach which is inspired from human market: the nodes trade their shares of applications' requests with others via auction and bidding to decide its own resource allocation and a global high-quality resource allocation is achieved as an emergent collective behavior of the market. Experimental results show that the proposed approach can ensure quick responsiveness, high scalability, and application prioritization in addition to managing the resources effectively.

Keywords distributed system, resource management, self-adaptation, shared cluster

1 Introduction

The emergence and popularization of new computing paradigms such as service computing[1], grid computing[2] and cloud computing[3], make it a cost-effective way for service providers to lease computing resources from third-party server providers under a "pay-as-you-go" model. At the server provider side, a large-scale cluster is maintained to host a large number of applications simultaneously. This kind of clusters is called shared clusters[4], where each application runs on a subset of nodes and these subsets can overlap with one another. In such a shared cluster, resource management includes two main tasks to satisfy the applications workloads and optimize the resource usage: *one is application placement by deciding which subset of nodes each application should run on; the other is workload assignment by deciding how many applications' workloads each node should be assigned.*

Due to the dynamics of application workloads[5] and the large scale of the shared cluster, manually managing the resource allocation is costly and even infeasible. Therefore, resource management is required to be self-adaptive to the workload changes. However, as the cluster scale and the application amount increase,adaptively managing the resource is becoming a crucial challenge: first, the management approach should response to the changes fast enough to avoid the circumstance where some applications are over-provisioned while the others are under-provisioned; second, the approach should be able to scale to a large number of nodes and applications; third, because the applications usually have different degrees of importance to the shared cluster, the approach should support application prioritization, that is, to allocate the high-priority applications with more resources in the case of resource confliction.

Most of existing resource management approaches[6-8] adopt a central controller to adjust current resource allocation based on the periodically collected global runtime information. However, their responsiveness is limited by the cycles of global information acquisition which may last very long in a large-scale cluster, and their scalability is constrained by the calculation of application placement which is an NP-hard problem[7]. Moreover, these approaches usually only use request satisfaction as the objective metric and make no prioritization among the applications when allocating resources.

Regular Paper

Supported by the National Basic Research 973 Program of China under Grant No. 2009CB320700, the National High Technology Research and Development 863 Program of China under Grant Nos. 2007AA010301, 2008AA01Z139, 2009AA01Z139-1, and the National Natural Science Foundation of China under Grant Nos. 60603038, 60773151.

*Corresponding Author

Currently, decentralized coordination techniques have been recognized as a promising solution to deal with the complexity and the dynamism in large-scale distributed systems[9-10]. Inspired from the human market coordination pattern[11-12], we propose a novel decentralized resource management approach in this article: 1) the cluster is modeled as a market, where the shares of application requests are treated as goods and nodes as dealers to exchange goods; 2) applications are assigned to different values to support application prioritization; each node sells or buys the request shares via auction and bidding to find a high-value collection of shares for itself; 3) the global resource management goal, which is to maximize the sum of the values of the satisfied application requests, is achieved as an emergent collective behavior of the market. Experimental results show that the proposed approach can ensure quick responsiveness, high scalability, and application prioritization in addition to managing the resources effectively.

The reminder of this paper is organized as follows. Section 2 introduces the related work; after that, Section 3 describes our market-inspired resource management approach in detail; Section 4 reports the experimental studies and analyzes the results; Section 5 discusses some issues about the proposed approach; finally, Section 6 concludes this paper.

2 Related Work

Many research efforts have focused on how to adaptively manage cluster resources among multiple applications. A typical approach is to adaptively partition cluster-wide servers into sub-groups and allocate each sub-group to a single application[13]. However, this approach does not allow the applications to overlap on the servers, while our approach allows multiple applications to share a same server to achieve finer granularity of cluster resources allocation and better resource utilization. A study[14] has pointed that fine-grained resource allocation at small time-scales can bring substantial multiplexing gains.

Most resource management approaches[6-8] for shared cluster employ a centralized architecture. Since application placement problem is a variant of NP-hard Class Constrained Multiple-Knapsack problem[7], these approaches dedicate to developing approximate algorithms such that high-quality solutions can be produced in a relatively short time. However, we believe such a centralized architecture has the following shortcomings: first, the controller depends on global runtime information acquisition, which cannot execute frequently in large-scale cluster, so these approaches' responsiveness to the workload variations is usually constrained; second, they are vulnerable to potential single point failure and scalability problem with the increase of applications and nodes; third, application prioritization is usually not supported in these algorithms.

To avoid the disadvantages of centralized resource management, Adam *et al.*[15] proposed a decentralized design. Similar with our approach, each node adaptively adjusts its resource allocation with the goal of optimizing its own utility. However, rather than adopting a negotiation mechanism, such as the auction in our approach, to coordinate the nodes' behaviors, their design lets each node adjust its own resource allocation independently which could make the entire cluster unstable and influence the approach's efficiency to some extent. Additionally, the convergence of the approach remains unclear.

Our approach borrows the idea from human market. Recently, some market-inspired resource management approaches have been proposed in grid computing (a survey can be found in [16]). These approaches concentrate on the batch workload. Usually it takes a relative long time to process a batch job, and as soon as the processing finishes, all the resources (e.g., memory) occupied will be released. The goal of these approaches is to determine the processing order for the coming batch jobs in order to satisfy each job's deadline. Different from these approaches, our approach focuses on the interactive workloads of web applications. This kind of workloads is usually short-running, and even if there is no request, the running web application still occupies a great amount of resources (e.g., memory). The goal of our approach is to determine the application placement among the nodes and the workload assignment among the application instances. Since these intrinsic differences, the designs of market mechanism are different. Typically, their market makes a "*price*" for the cluster's resources based on some factors, such as current cluster's workloads, available resources; and then the jobs' owners offer their own "*bid*" according to current "*price*" and the jobs' requirements, such as job deadline, emergence; finally, cluster finds an appropriate processing order, which satisfies the jobs' requirements as well as maximizes the values acquired by the cluster, in virtue of some market mechanisms.

Essentially, load balancing associates applications' workloads with the cluster's resource in order to optimize the resource usage in the cluster. However, existing load balancing approaches[17-18] usually focus on how to dispatch the workloads to the nodes with fixed application instances. Hence they do not care about how to place the application instances among the nodes, which is an importance concern of our approach.

3 Self-Adaptive Resource Management Approach

In this section, we first give a formal statement of the resource management problem; after that, we give an overview of our market-inspired approach, and introduce three key design issues in details; and then, we proof the convergence of the approach; finally, we present our prototype implementation.

3.1 Problem Statement

Fig.1 gives a snapshot of the shared cluster, where all the nodes are connected by a high-throughput low-latency network. The *dispatcher* forwards every application request to one of the application's running instance based on its dispatching strategy. Each node runs an application server (AS), which provides runtime environment for the running application instances.

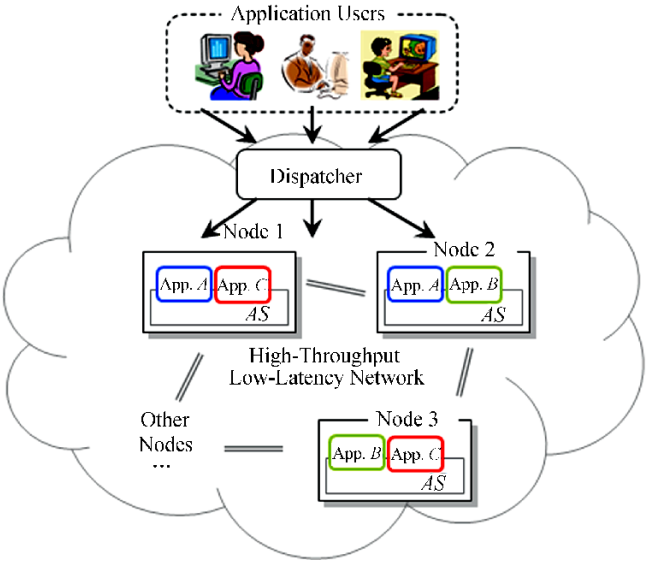

Fig.1. A typical shared cluster.

Our current approach mainly focuses on the workloads of web applications whose requests are typically processed at a high rate, and the resources of business logic layer. The resources that our approach deals with are CPU and memory, and our approach assumes that nodes have different amount of these resources. As pointed by previous work[6], the CPU consumption primarily increases along with the request rate, whereas the memory consumption of a running application instance holds in a stable range even if the application instance receives no requests. Consequently, in this paper, we assume that an application's request rate and average CPU cycles to process a request can be estimated at runtime, and an upper limit for memory consumption of an application instance is estimated beforehand.

To support application prioritization, the cluster provider can assign a value to a request of each application, and the larger the value, the higher the priority. Accordingly, the goal of resource management is to maximize the sum of values acquired by satisfying the requests in the shared cluster under the resource constraints, which is formally defined below.

Definition 1. *Let* $Sum_{\text{value}} = \sum_{n\in N}\sum_{a\in A_n} \vartheta_{n,a} \cdot \nu_a$, *such that*:

$$\forall n \in N, \quad \sum_{a\in A_n} \vartheta_{n,a} \cdot c_a \leqslant C_n, \quad \sum_{A\in A_n} m_a \leqslant M_n.$$

The resource management goal is:

$$\max(Sum_{\text{value}})$$

where:

- a $(a \in A)$ *denotes an application,* n $(n \in N)$ *denotes a node in the shared cluster respectively*;
- ν_a, c_a *and* m_a *denote the utility value, the average CPU cycles to process a request, and the required memory of an instance of application* a, *respectively*;
- $\vartheta_{n,a}$ *denotes the satisfied requests of application* a *on node* n, *and* A_n *is the set of applications hosted by node* n;
- M_n, C_n *denote the memory and the CPU capacity of node* n *respectively.*

When the values of all the applications are the same, the problem is reduced to maximize the amount of satisfied requests, which is the goal of most previous studies[6-7].

3.2 Approach Overview

For the term "market", on its most basis, it means a place where goods are bought and sold among several dealers according to their willingness. The key idea of our market-inspired approach can be summarized as follows.

1) To model the shared cluster as a market, we treat nodes as "dealers" and the shares of the requests of the applications (referred as *application shares* or *shares*) as "goods". By shares, we mean if a node has 20% shares of an application, it will receive 20% of the requests of the application. To support this idea, the dispatcher is designed to dispatch application requests according to the proportion of shareholdings of the nodes.

2) Further, based on local estimation of the application workloads, each node valuates the application shares and trades with others autonomously in order to get a collection of shares, which fits the node's resource constraints and maximizes its *income* (refer to Subsection 3.3).

3) Through a series of decentralized and self-organizing trades, the shared cluster converges toward a global high-quality allocation of goods when dealers gradually stabilize with their own satisfactory collections of shares.

To illustrate the idea clearly, we give an example in Fig.2. In the shared cluster, suppose application C's request rate suddenly surges; as a result, node 1 is unable to process all the incoming requests of application C due to the limited CPU resource (Fig.2(a)); the colored rectangle represents the amount of application C's workload on a node. Under this circumstance, node 1 decides to sell 50% shares of application C and then multicasts an auction to some other nodes (see Fig.2(b), step 1). After hearing of the sale message, each buyer first makes a valuation itself; in this example, node 50 prefers to buy 35% shares at the sacrifice of uninstalling application B since application B's workload has slowed down for a while; node 65 prefers to buy 20% shares with its vacated CPU; node 100 has no interest in buying because this requires it to give up some more valuable requests from application A. Hence, node 50 and node 65 reply node 1 with a bid message respectively and wait for the result (see Fig.2(b), step 2). By sorting all the received bids (see Fig.2(b), step 3), node 1 decides to satisfy node 65's purchase demand (20% shares) first and sell the rest (30% shares) to node 50. And then, it notifies the buyers and the dispatcher (see Fig.2(b), step 4) of the results respectively.

After receiving the update message, the dispatcher changes the workload assignment accordingly (see Fig.2(c)). Moreover, due to the uninstallation, node 50 will continue to start a new auction to sell its unsatisfied shares of application B after this trade.

Based on this approach, the global resource management problem is decomposed into each node's local problem, which can be solved quickly as its scale holds small and stable. As trades incrementally propagate over the shared cluster, resource re-allocation for the dynamic workload changes is achieved rapidly in a decentralized manner.

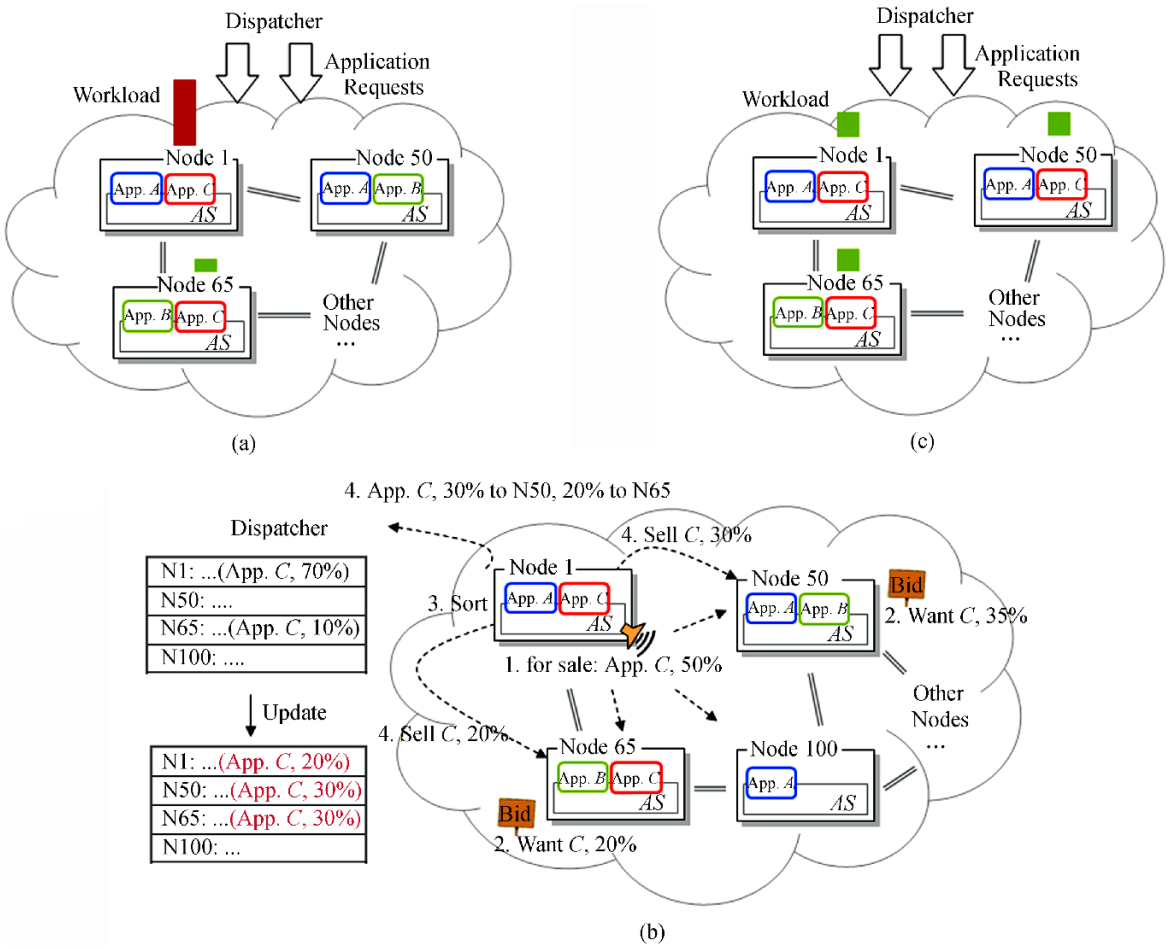

Fig.2. An example. (a) The initial status. (b) The trading process. (c) After the trade.

3.3 Key Design Issues

There are three key issues in the design of our approach: first is the income, which acts as an incentive for the dealers to obtain or relinquish goods when doing so promises an income growth; second is a dealer's local goods valuation strategy, which defines how it chooses the goods for sale or to buy; third is the trade mechanism, which is to determine the goods allocation when multiple buyers compete for the same goods.

3.3.1 Income Definition

The "income" of a node is calculated as follows.

Formula 1. *Formally, let* $\gamma_{a,n}^{\text{served}}$ *and* $\gamma_{a,n}^{\text{dropped}}$ *be the amount of the requests of application a served and dropped by node n respectively; then,*

$$income_n = \sum \gamma_{a,n}^{\text{served}} \cdot \nu_a - \sum \gamma_{a,n}^{\text{dropped}} \cdot \nu_a. \quad (1)$$

This formula represents that a node obtains income by processing requests, and loses income because of dropping requests due to a lack of resources for processing requests. Accordingly, to maximize its income, a node is stimulated to sell out parts of its shares when its resources are not enough, and to buy in more shares when its resources are idle.

3.3.2 Node's Local Valuation Strategy

When executing trading behaviors, a node decides which to sell according to its *selling valuation strategy*, and which to buy according to its *buying valuation strategy*. In both valuation strategies, memory and CPU requirements of each application are taken into consideration. These resource requirements are estimated at runtime by the node (for some details of the estimation, please refer to Subsection 3.5).

Selling Valuation Strategy. The goal is to find a high-quality shares-collection that fits into the node's memory and CPU constraints and brings high income, and then to sell all the other shares excluded from the collection. When doing the selling valuation, the node abstracts the problem as a 0-1 Knapsack Problem[19], which is to find the best collection of items that fits into one bag and maximize the bag's value, given each item a weight, a volume and a value. The time complexity of the 0-1 Knapsack algorithm is $O(n^2)$ where n is the number of candidate items, so to achieve high valuation efficiency, the node considers the shares of one application as a whole item during the selling valuation, in which way the number of candidate items is reduced to the number of applications on the node. Further, as the number of applications on each node is very small (usually no more than 5), the node can traverse all the possible collections very quickly even though the time complexity of the algorithm is high.

Algorithm 1. Find a High-Quality Collection of Shares

Inputs: an array of share items (*shares*).

```
1.  Initialize
    k = 0; stack = an empty stack;
    collection = null; //the high-quality collection
    income_max = 0; //the income of collection
    mem = node's memory; cpu = node's CPU;
2.  Do
3.      While (k < shares.length && mem > 0 &&
        cpu > 0)
4.          If (mem >= shares[k].memory)
5.              mem = mem − shares[k].memory;
6.              Clone shares[k] to a new item of shares;
7.              If (cpu >= shares[k].cpu)
8.                  cpu = cpu − shares[k].cpu;
9.              Else
10.                 item.cpu = cpu;
11.                 cpu = 0
12.             Push the item into stack;
13.         k++; //try the next application shares
14.     Calculate the income of collection in the stack;
15.     If (income > income_max)
16.         income_max = income;
17.         Copy the items in the stack to collection;
18.     If (stack is not empty)
19.         Pop the top shares item from stack;
20.         mem = mem + shares[item.app].memory;
21.         cpu = cpu + item.cpu;
22.         k++;
23. While (k != shares.length && stack is not empty)
24. Return collection;
```

For the shares of one application, we design a data structure to record its memory (item.memory) and CPU (item.cpu) requirements, and the identifier of the application (item.app). Then, the node finds the high-quality collection of shares as follows (Algorithm 1): the node finds a collection of shares by adding the candidate items of shares one by one until either its resources are not enough or there is no more items (lines 3∼13). One thing to be mentioned is that if a new added item leads to the shortage of the node's CPU, the algorithm will cut the excess shares from the item (lines 9∼11). After that, the node calculates the income of the found collection according to (1), and if it is larger than the current maximal income, the node records the income and the collection respectively (lines 14∼17). Then,

the node removes one item of shares from the current collection and recovers its state (lines 18∼22) in order to look for the next possible collection. Finally, the node returns the highest-quality collection it found.

Buying Valuation Strategy. Though the received sale messages may contain shares of multiple applications, we assume that a node bids for at most one application's shares in a specific trade. Accordingly, the goal is to find the shares of an application that bring the maximal *profit* (see Formula 2) to the node.

Formula 2. *Formally, for the shares of one application on sale, let* $income_{\text{new}}$ *and* $income_{\text{cur}}$ *denote the node's income after buying the shares and the node's current income, respectively. Let* $cost_{\text{uninstall}}$ *be the average cost of uninstalling an application, and* n *be the number of applications to be uninstalled for buying the new shares. Then,*

$$profit = income_{\text{new}} - income_{\text{cur}} - n \cdot cost_{\text{uninstall}}. \quad (2)$$

Considering the node's memory constraint, a node buying some shares of an application may result in uninstalling other applications, which brings extra cost and disturbance to the shared cluster, so in Formula 2 we introduce the $cost_{\text{uninstall}}$ variable to reduce the profit the node gets from buying the shares, and further to reduce the node's desire to buy. Moreover, since the bid a node offers is in proportions to the profit (see Section 3), the seller will be more likely to sell the shares to the nodes that have already installed the application, and finally the times of uninstallations in the shared cluster is reduced. The value of $cost_{\text{uninstall}}$ is set by the cluster administrator. The smaller the value is, the more frequently a node changes its hosting applications for a higher income. Accordingly, the cluster administrator can make a trade-off between system stability and system adaptability by assigning the parameter different values.

Algorithm 2. Finding the Most Profitable Shares on Sale

Inputs: the array of share items on sale (*sharesOnSale*) the array of node's current share items (*curShares*)

1. **Initialize**
   $income_{\text{cur}}$ = the current income of the node;
   $profit_{\max} = 0$, $amount_{\text{tobuy}} = 0$, $app_{\text{tobuy}} = -1$;
2. **For** each *sharesOnSale*[*i*]
3. **Merge** *sharesOnSale*[*i*] into *curShares*
4. *collection* =
5. **FindHQShareCollection** (*curShares*);
6. **If** (*collection*.contains(*sharesOnSale*[*i*].app))
7. **Calculate** the *profit*
8. **If** (*profit* > $profit_{\max}$)
9. **Reset** $amount_{\text{tobuy}}$ and app_{tobuy};
10. $profit_{\max}$ = *profit*;
11. **Remove** *sharesOnSale*[*i*] from *curShares*;
12. **Return** $amount_{\text{tobuy}}$ and app_{tobuy};

Algorithm 2 valuates the items of shares on sale as follows: the FindHQShareCollection function (line 5) is Algorithm 1; after valuating the shares on sale one by one, if there is something worth buying (i.e., $profit_{\max} > 0$), the algorithm returns the application identifier and the shares amount that the node should to buy.

3.3.3 Trading Mechanism

An auction-based trade mechanism is designed to make the seller sell its goods to the appropriate buyers. Fig.3 depicts the four consecutive steps in a trade.

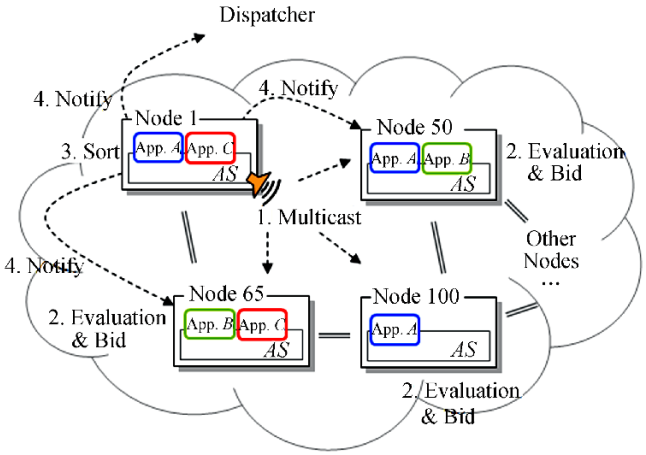

Fig.3. Four consecutive steps in a trade.

First, a node (called seller) multicasts sale messages to a small subset of nodes (called buyers), which are chosen randomly, and waits for responses. The sale message contains the application identifiers, shares for sale, the estimated request rates and average CPU cycles for processing a request, and the period of validity of the auction.

Second, each buyer independently valuates the received message. If the buyer finds something profitable to buy, it replies to the seller with a bid message. The bid message tells the identifier and the shares of the application the buyer wants to buy, and also the bid the buyer offers. Then, the buyer waits for the bidding result. The bid, calculated by (3), indicates the buyer's purchase willingness. Similar with human market, the price is in direct proportion with the profit the buyer gets. Besides, considering the less memory the buyer left, the less possibility it installs a new application instance, the buyer would like to offer higher bid to fully

utilize its CPU.

Formula 3. *Formally, for node n, let profit be the profit of buying the shares; let $\Gamma_n^{\text{residual}}/\Gamma_n$ denote its idle memory ratio, where Γ_n and $\Gamma_n^{\text{residual}}$ are its total and residual memory respectively; then,*

$$bid_n = \begin{cases} \dfrac{1}{\Gamma_n^{\text{residual}}/\Gamma_n} \cdot profit, & \Gamma_n^{\text{residual}} \neq 0, \\ \alpha \cdot profit, & \Gamma_n^{\text{residual}} = 0. \end{cases} \tag{3}$$

As shown in (3), when $\Gamma_n^{\text{residual}}$ is zero, we enlarge the influence of *profit* by multiplying it by a constant α which is designated to 1000 in our current implementation.

Third, the seller determines how many shares each buyer wins according to their bids: the shares are sold to the buyers in a bid ascending order until all are sold out or there is no buyer.

Finally, the seller notifies the buyers of the bidding results and the dispatcher of the share distribution changes. After receiving the result, if buyers need to install a new application, it will first fetch the binary code of the application from the application repository, and then install the application.

3.4 Convergence of the Approach

In the shared cluster, each node will continuously adjust its own resource allocation among the applications' workloads through trades to maximize its income. After each trade, the requests processed by the seller remain unchanged, and parts of the requests unprocessed by seller are transferred to the buyers; a buyer accepts the requests only when the new requests can bring it some profit. In other words, the sum of values acquired by satisfying the requests in the shared cluster (i.e., Sum_{value} in Definition 1) is increased by at least the value of one request after a trade. On the other hand, Sum_{value} of the shared cluster is bounded at a moment. Therefore, after finite times of trades, the shared cluster inevitably converges into a stable status. A formal proof is illustrated below.

Theorem 1 (Convergence). *At a specific moment, let $N = \{n\}$ and $A = \{a\}$ be the set of nodes and the set of applications in the shared cluster, respectively; let $T = \{t\}$ be the set of trades occurred after the moment. If the workloads of all the application keep stable, then T must be a finite set.*

Proof. From Definition 1, we have:

$$\forall n,\ income_n = \sum \gamma_{a,n}^{\text{served}} \cdot \nu_a - \sum \gamma_{a,n}^{\text{dropped}} \cdot \nu_a.$$

Let I be the sum of all the nodes' incomes, that is,

$$I = \sum_{n \in N} income_n = \sum \gamma_a^{\text{served}} \cdot \nu_a - \sum \gamma_a^{\text{dropped}} \cdot \nu_a$$

$\forall a \in A$, θ_a ($\theta_a \geqslant 0$) be the number of application a's request received by the shared cluster, then we have:

$$-\sum_{a \in A} \vartheta_a \cdot \nu_a \leqslant I \leqslant \sum_{a \in A} \vartheta_a \cdot \nu_a. \tag{4}$$

Let $t_{p,q}$ be a trade in which node n_p sells some requests to node n_q, $t_{p,q} \in A$. Let ΔI_t be the increment of income brought by trade t, then,

$$\forall t_{p,q},\ \Delta I_{t_{p,q}} = \Delta income_{n_p} + \Delta income_{n_q}.$$

In a trade, for the seller, the requests processed keep unchanged while the requests dropped decrease at least one, so $income_{n_p}$ increases at least $v_{\min}$, and here $v_{\min}$ denotes the minimal value of a request in the shared cluster. That is,

$$\Delta income_{n_p} > v_{\min}.$$

For the buyer, as stated by the buying valuation strategy, its profit acquired must be greater than zero, i.e.,

$$profit_{n_q} = \Delta income_{n_q} - n \cdot cost_{\text{uninstall}} > 0$$
$$\Delta income_{n_q} > 0.$$

Then we have:

$$\Delta I_{t_{p,q}} > v_{\min}. \tag{5}$$

Further, it will not hurt to suppose trades happen one by one; let t_i be the i-th trade, and I_i be the sum of nodes' incomes after the i-th trade; based on (5), we have:

$$I_2 - I_1 > v_{\min},\ I_3 - I_2 > v_{\min}, \ldots, I_n - I_{n-1} > v_{\min}.$$

Add the two sides of the inequality respectively, then we get:

$$I_n - I_1 > n \cdot v_{\min}. \tag{6}$$

Based on (4), we have:

$$-\sum_{a \in A} \vartheta_a \cdot \nu_a \leqslant I_1 \leqslant \sum_{a \in A} \vartheta_a \cdot \nu_a,$$
$$-\sum_{a \in A} \vartheta_a \cdot \nu_a \leqslant I_n \leqslant \sum_{a \in A} \vartheta_a \cdot \nu_a.$$

Then we get

$$-2\sum_{a \in A} \vartheta_a \cdot \nu_a \leqslant I_n - I_1 \leqslant 2\sum_{a \in A} \vartheta_a \cdot \nu_a. \tag{7}$$

Based on (6), (7), we have:

$$n \cdot \nu_{\min} \leqslant 2\sum_{a \in A} \vartheta_a \cdot \nu_a.$$

Then we get

$$n \leqslant \frac{\sum_{a \in A} 2\vartheta_a \cdot \nu_a}{v_{\min}}.$$

Consequently, the times of trades, n, are finite; the convergence of the proposed algorithm has been proved. □

3.5 Prototype Implementation

As shown in Fig.4, our prototype contains the following main components in the shared cluster: the *dispatcher* receives application requests and forwards them to the nodes; the *application server* running on each node implements the node's local resource allocation logic; the *repository* stores the binary codes of applications. When deploying an application, the node first searches its local repository, and if the application is not found there, the node can download it from a remote repository which are placed in the shared cluster; the *management console* provides management functionalities for system administrators to configure system parameters (e.g., uninstallation cost), and to add or remove an application from the shared cluster.

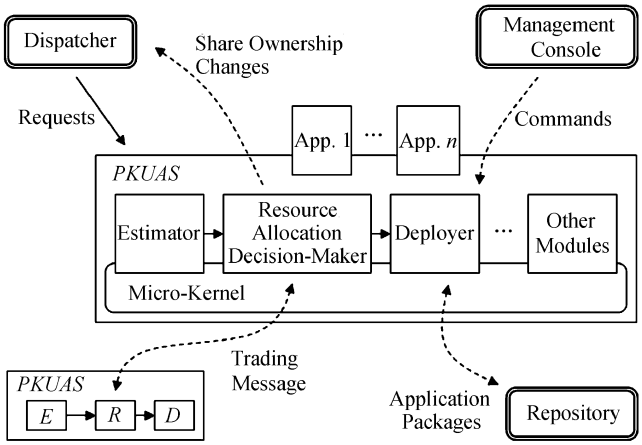

Fig.4. The resource management support on the PKUAS application server.

The dispatcher maintains a table to record the proportions of shareholdings of the nodes, and updates the table according to the seller's notification at the end of each trade. When receiving an application request, the dispatcher forwards it to a node in accordance with certain probabilities, i.e., the shareholdings.

The application server adopted is PKUAS (PeKing University Application Server)[20], which has a highly extensible architecture. To support our approach, we extend PKUAS with three modules: the *estimator* module periodically estimates each hosted application's request rate and its average CPU cycles for processing a request with an existing technique[21]. Accordingly, the CPU requirements of an application's shares can be estimated. Based on the estimation data, the *resource allocation decision-maker* module evaluates the node's load status and enters the behavior mode (introduced in the next paragraph) accordingly. Once the node decides to change its application placement, the *deployer* module is invoked to install and uninstall the applications: when installing an application, the deployer automatically retrieves the application's code from a repository, and deploys it on the application server.

We define three behavior modes, which are *buy-dominant* mode, *sell-dominant* mode, and *buy-sell-alternate* mode for a node to switch periodically. Fig.5 describes the behavior mode transition. In each mode, the node tries to buy or sell many times until it either buys or sells something, or this mode's period is over. Each time when a node tries to buy, it will first parse the sale messages coming from other nodes, and evaluate the goods on sale, and join a trade only when it finds something profitable. Meanwhile, each time when a node tries to sell, it will evaluate its current applications' shares, and then initial a trade only if it finds some shares unable to process. Different behavior modes have different proportions of buying times to selling times. Take buy-dominant mode for example, if the decision-maker finds the node's resource usage ratio is lower than a threshold, that is, most of the node's resource are idle, the node will come into the buy-dominant mode where the times of trying to buy is more than that of trying to sell (the node may try to buy twice, and then try to sell once).

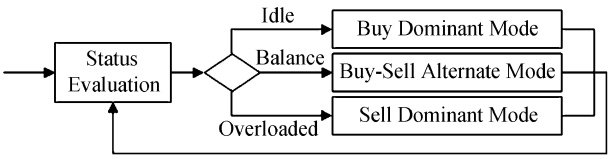

Fig.5. Behavior changes of a node.

Each node preserves some global information in the trading process. When selling shares, the node uses a list of the cluster's nodes to randomly pick up a small subset of nodes as the buyers (in our current implementation, the number is 20). When valuating goods, the node uses some system-wide parameters, such as the utility value of an application and the uninstallation cost. Updating such information is based on a system-wide broadcasting from the management console.

4 Evaluation

In this section, we evaluate the effectiveness, responsiveness, scalability and application prioritization support of our approach with a set of simulation experiments.

4.1 Experimental Settings

In the experiments, we simulate each node of the shared cluster with three threads. The first thread is used for request processing: it periodically retrieves application requests from the node's local request queue in a priority ascending order and then processes them. The nodes can process the requests only if it has installed the application and its CPU is not used out. If the total CPU demand of the requests in the queue exceeds the CPU resource offered by the node in one cycle, the left requests will be discarded. The second thread is used for trade executing: it periodically evaluates the node's current requests processing status, and executes buying or selling behaviors according to the modes described in Fig.5. The third thread is used for message processing: on one hand, it sends sale messages, bid messages, and bidding result messages to other nodes; on the other hand, it accepts the messages from other nodes and putts the sale messages and bid messages into different buffers respectively. Nodes communicate with each other via sockets; they adopt UDP protocol to send the sale messages and the bid messages, and adopt TCP protocol to send the bidding result messages.

Meanwhile, we use a process to simulate a dispatcher. The simulated dispatcher generates the applications' workloads per second according to the experiment requirements, and then forwards the requests to the nodes according to current proportions of shareholdings of the nodes.

We program the simulated experiment in Java. The experimental environment includes eleven machines interconnected by a dedicated 100 Mbps Ethernet LAN. Each machine is equipped with a 2.0 GHz processor and 1 GB RAM. The operating system is Microsoft Windows XP professional, service pack 3. In the experiments, one machine is designated to host the dispatcher, while the others are used to host the simulated nodes. The nodes on the same machine use different network ports and function independently. Each machine hosts 100 nodes simultaneously, so the experiments can run up to 1000 nodes.

The nodes' memory and CPU configurations are uniformly distributed over the set {1 GB: 1 GHz, 2 GB: 1.6 GHz, 3 GB: 2.4 GHz, 4 GB: 3 GHz}, which are borrowed from Tang's work[7]. And without lack of generality, we assume that all the nodes have homogeneous CPU architectures (i.e., processing a same request requires same CPU cycles on each node). According to the characteristics of typical web applications, we assume that a request of each application has the same average execution time of 250 ms, and its execution time follows an exponential distribution with $\lambda = 4\,\mathrm{s}^{-1}$ on each node. The applications' memory requirements are uniformly distributed over the set {0.4 GB, 0.8 GB, 1.2 GB, 1.6 GB}. To represent the applications' CPU requirements, we introduce CPU Load Factor (lf_{cpu}) to represent the intensity of the external workloads, which is a ratio between applications' total CPU requirement and the cluster's total CPU capacity. When $lf_{\mathrm{cpu}} = 1$, it means the shared cluster is saturated. Once lf_{cpu} and the data center's total CPU capacity are settled, the total application CPU requirement (i.e., the total workloads) is also determined.

All the experiments are run 10 times respectively, so each reported data point in the figures is an average of 10 results. Based on different evaluation goals, more details of the experimental settings are given in the following subsections.

4.2 Effectiveness Under Dynamic Workloads

In the experiment, we evaluate the approach's effectiveness and responsiveness under dynamic changing workloads. The shared cluster under evaluation has 200 nodes and 100 applications whose values are the same. Initially, the total workloads are partitioned among the applications with a power-law distribution, where the workload proportion of the i-th application is set to $i^{-2.16}$, and the application placement and the workload assignment among the nodes are generated randomly. At runtime, the lf_{cpu} is set to 1, and all the applications' workloads vary randomly within ±50% every 10 seconds.

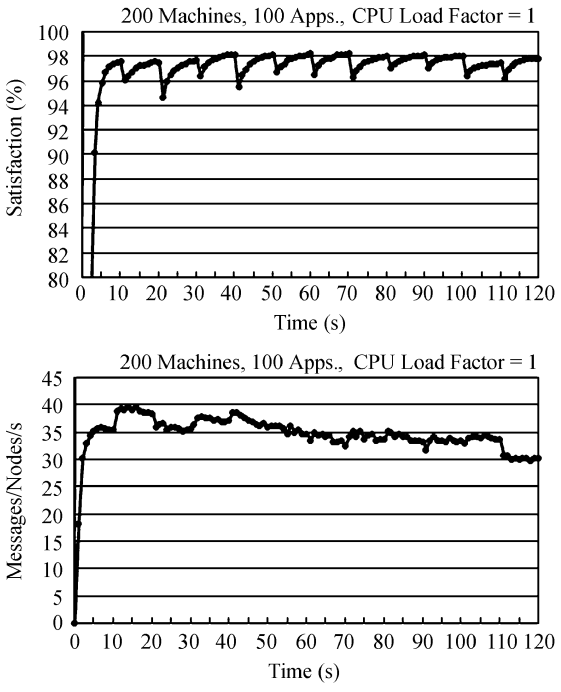

Fig.6. Effectiveness of the approach under dynamic workloads.

As mentioned in Subsection 3.1, resource management goal is simplified to maximize the total satisfied requests when all the applications have the same value. Fig.6 shows that the request satisfaction, computed as the ratio between the satisfied requests and the total requests in the shared cluster, rapidly converges at a high value from a random initialization. Each time when the applications' workloads change, the cluster makes a rapid response, and then the request satisfaction recovers to a high value again. Besides, we recorded the message overhead introduced by our approach (35 messages per second per node on average). We believe the overhead is affordable, because these messages are very small (less than 10 bytes) and can be processed quickly.

4.3 Effectiveness Under Different Workloads

In the experiment, we evaluate the approach's effectiveness under different workloads of the data center. To generate different external workloads of the data center, we vary lf_{cpu} from 0.2 to 2.0. The data center under evaluation has 200 nodes and 100 applications whose values are the same. The initialization of the application workloads is similar with the previous experiment. For each lf_{cpu}, after the data center enters a stable status, a free workload variation, which means that all the applications' workloads vary freely and independently, occurs at an arbitrary time. When the data center enters a new stable status again, we record the execution time for finding the new solution. We find that the peak value of the time is below 25 seconds, which is affordable (see Fig.7); besides, we also recorded the resource utilization of data center (abbr. *utilization*), which is computed as the ratio between the resources used and the total resources of the data center. As shown in Fig.7, *utilization* increases linearly with $lf_{\text{cpu}} < 1$, and stands above 96% when $lf_{\text{cpu}} > 1$.

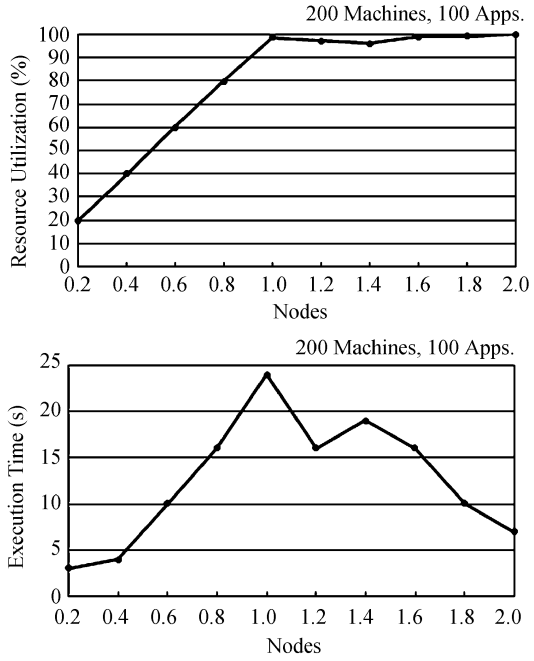

Fig.7 Effectiveness of the approach under different workloads.

4.4 Scalability

In the experiment, we evaluate the approach's scalability by varying the data center's scale from 100 nodes, 50 applications to 1000 nodes, 500 applications. The initialization of the application workloads is similar with the previous experiment and lf_{cpu} is set to 1 at runtime. Similar with previous experiment, for each scale, a free workload variation occurs at an arbitrary time, and we recorded the execution time for finding the new solution.

As shown in Fig.8, the execution time increases linearly with the scale, and the achieved satisfaction holds above a high value. Besides, we found the average message overhead for a node gradually decreases with the data center scaling up.

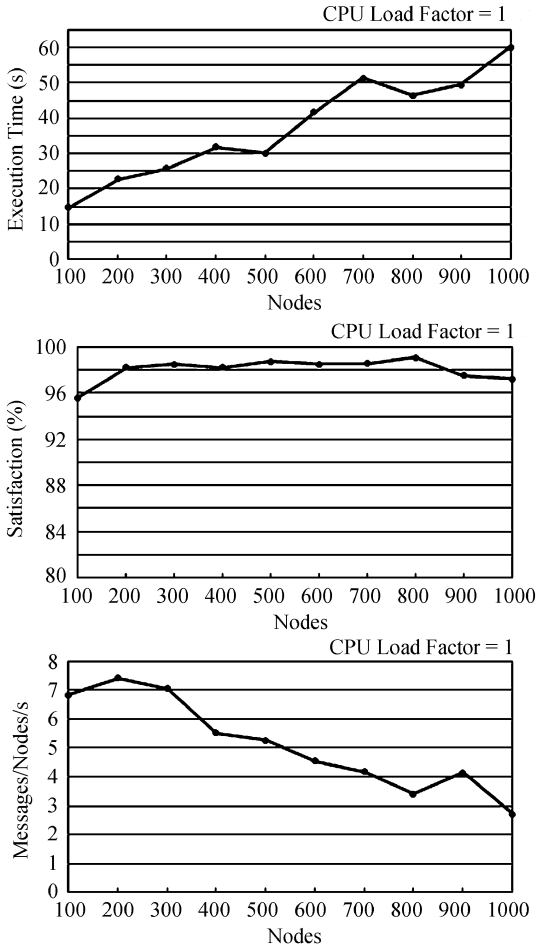

Fig.8. Scalability of the approach.

4.5 Application Prioritization

In the experiment, we evaluate the application prioritization support of our approach. The cluster under evaluation has 200 nodes and 100 applications. We assume the applications are divided into three categories, whose priorities are gold (highest), silver and bronze respectively. Initially, the CPU demands of the three categories are in the proportions of $1:3:6$, and lf_{cpu} is set to 1. At a certain time (the 5th second in Fig.9) a request spike of silver applications is simulated: the CPU demands of silver application increase from 30% to 50% of original total demands.

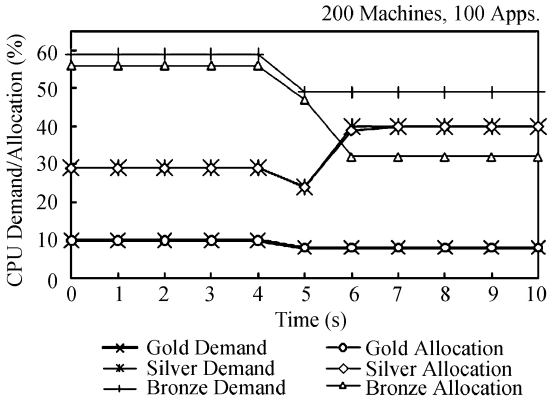

Fig.9. Application prioritization support of the approach.

Since the shared cluster's CPU capacity has been saturated, we observed that the approach responded to the request spike by allocating more CPU cycles to the silver applications at the sacrifice of dropping some bronze requests, while holding the CPU cycles allocated to the gold applications stable (see Fig.9). In other words, our approach can adaptively allocate resources to the applications with high priority.

5 Discussion

The central idea of our approach is to find a global satisfactory resource allocation via the nodes' local adaptation and interaction behaviors, which take little time as the problem scale holds small and stable. Although the resource allocation we get may not be optimal, we believe it is essential for the cluster to find a high-quality solution in a short time rather than to spend much time searching for a temporary optimal one. This is because that finding the theoretical optimal resource allocation is NP-hard, and the optimal solution changes continuously due to the ever-changing application workloads.

Intuitively, one might have considered the dispatcher as another central component in our solution, since the cluster still depends on the dispatcher to forward the requests. However, note that the probability-based dispatching strategy nearly takes no time, which makes the dispatcher forward request very fast; moreover, it is feasible to place a group of dispatchers working simultaneously with an in-group synchronization mechanism to update the application share ownership changes. Therefore, we believe the dispatcher will not be a potential performance bottleneck of our approach.

Although the goods valuation and trading process are decentralized, the nodes rely on some global information, whose update is based on broadcasting. However, the update rarely happens because the information is relative stable, and the update message is small (around 50 bytes). Therefore, maintaining such information has little impact on the cluster's performance in practice.

The time needed for a node to load a new application contains two parts: the time needed to download the application and the time needed to install it. Since the shared cluster are typically connected by high-speed network and nodes can download some popular application to their local disks in advance, the time needed to download the application can be so short to be ignored. As for the application installation time, our ongoing work is to integrate the live migration technique of virtual machine into the current implementation, and thus the installation is simplified as loading a well-configured application image, whose time cost can be reduced to millisecond order of magnitude based on existing techniques[22].

In our current approach, we have not considered the influence of application server's cache mechanism, which is used to keep some response results for improving the response speed of the follow-up requests. However, to make our approach cache-aware, some improvements could be done in the node's buying and selling evaluation algorithms. For example, if the results of application a have been cached on node n, node n can enlarge $\sum \gamma_{a,n}^{\text{served}} \cdot \nu_a$ to increase the probability of preserving application a. An in-depth study is being considered in our future work.

One limitation of our approach is that the current design does not support CPU utilization balancing across the nodes. A feasible solution may be to allow a node to buy application shares proactively when its resource utilization is lower than some other nodes. In our ongoing work, we are extending our current implementation to support this feature.

There are two reasons for us to evaluate the proposed approach in a simulated environment. First, in a practical cluster system, the request satisfaction achieved will not only depend on our proposed approach, but also be influenced by other system parts. For example, the accuracy of the workload estimation method used

in the cluster will impact the correctness of a node's buying and selling decisions, and further influence the performance of the whole cluster. Using simulated system components can help eliminate the influence of other system components and focus on evaluating the proposed approach. Second, the machines we can use in our experiment are also limited. Therefore, it is not possible for us to evaluate our approach in real cluster system until now. As the overhead of our approach roots in the communication and local valuation of the nodes, we believe our simulation environment can validate our approach well because of the following reasons: first, the communication in the simulated nodes is implemented via sockets, which is the same as in a real cluster, that is, the overhead brought by communication has been considered; second, in the simulation, a physical machine is used to simulate up to 100 nodes, that is, the CPU that each node uses to execute the local valuation is no more than 1% of a physical machine's CPU, which is also affordable in a real cluster.

6 Conclusion

In this paper, we proposed a self-adaptive resource management approach for shared clusters. Inspired by human market, the approach models the shared cluster as a market: each node trades application shares with others to decide its own resource allocation, and all the trades make the entire cluster converge toward a global high-quality resources allocation. The key design issues of our approach, including the income calculation, the local goods valuation strategy of dealers, and the trade mechanism, were explained in depth. Finally, we evaluate the proposed approach with respect to its effectiveness under dynamic changing application workloads and different workloads, its scalability, and its ability to prioritize applications.

Acknowledgements We would like to thank Xuan-Zhe Liu for his valuable advice and the members of application server group from Institute of Software, Peking University.

References

[1] Tsai W T. Service-oriented system engineering: A new paradigm. In *Proc. IEEE International Workshop on Service-Oriented System Engineering* (*SOSE*), Beijing, China, Oct. 20-21, 2005, pp.3-6.

[2] Foster I, Kesselman C (eds.). The Grid: Blueprint for a New Computing Infrastructure. San Francisco: Morgan Kaufmann Publishers Inc., 1999, p.677.

[3] Armbrust M, Fox A, Griffith R, Joseph A, Katz R, Konwinski A, Lee G. Patterson D, Rabkin A, Stoica I, Zaharia M. Above the clouds: A Berkeley view of cloud computing. University of California, Berkeley, Tech. Rep., 2009, http://d1smfj0g31qzek.cloudfront.net/abovetheclouds.pdf.

[4] Urgaonkar B, Shenoy P, Roscoe T. Resource overbooking and application profiling in shared hosting platforms. In *Proc. Fifth Symposium on Operating Systems Design and Implementation*, Boston, USA, Dec. 9-11, 2002, pp.239-254.

[5] Crovella M E, Bestavros A. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Trans. Networking*, 1997, 5(6): 835-846.

[6] Karve A, Kimbrel T, Pacifici G, Spreitzer M, Steinder M, Sviridenko M, Tantawi A. Dynamic application placement for clustered Web applications. In *Proc. the International World Wide Web Conference*, Edinburgh, UK, May 23-26, 2006, pp.595-604.

[7] Tang C, Steinder M, Spreitzer M, Pacifici G. A scalable application placement controller for enterprise data centers. In *Proc. the International World Wide Web Conference*, Banff, Canada, May 8-12, 2007, pp.331-340.

[8] Rolia J, Anderzejak A, Arlitt M. Automating enterprise application placement in resource utilities. In *Proc. the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management* (*DSOM*), 2003, Heidelberg, Germany, Oct. 20-22, pp.118-129.

[9] Tan W, Fan Y. Decentralized workflow execution for virtual enterprises in grid environment. In *Proc. the 5th International Conference on Grid and Cooperative Computing Workshops* (*GCCW 2006*), Changsha, China, 2006, pp.308-314.

[10] Andrzejak A, Graupner S, Kotov V, Trinks H. Algorithms for self-organization and adaptive service placement in dynamic distributed systems. Technical Report HPL 2002-259, Hewlett Packard Labs Palo Alto, 2002.

[11] Clearwater S H (ed.). Market-Based Control: A Paradigm for Distributed Resource Allocation. Singapore: World Scientific, 1996.

[12] De Wolf T, Holvoet T. Design patterns for decentralised coordination in self-organising emergent systems. In *Proc. the 4th International Workshop on Engineering Self-Organing Systems* (*ESOA*), Hakodate, Japan, May 9, 2006, pp.28-49.

[13] Appleby K, Fakhouri S, Fong L, Goldszmidt G, Kalantar M, Krishnakumar S, Pazel D, Pershing J, Rochwerger B. Oceano: SLA based management of a computing utility. In *Proc. International Symposium on Integrated Network Management*, Seattle, USA, May 24-25, 2001, pp.14-18.

[14] Chandra A, Goyal P, Shenoy P. Quantifying the benefits of resource multiplexing in on-demand data centers. In *the 1st ACM Workshop on Algorithms and Architectures for Self-Managing Systems*, San Diego, USA, Jun. 11, 2003.

[15] Adam C, Stadler R. Service middleware for self-managing large-scale systems. *IEEE Trans. Network And Service Management*, 2007, 4(3): 50-64.

[16] Broberg J, Venugopal S, Buyya R. Market-oriented grids and utility computing: The state-of-the-art and future directions. 2007, http://gridbus.org/reports/MarketGridUtilitySurvey2007.pdf.

[17] Teo Y, Ayani R. Comparison of load balancing strategies on cluster-based web servers. *The Journal of the Society for Modeling and Simulation International*, 2001, 77(5/6): 185-195.

[18] Ungureanu V, Melamed B, Katehakis M. Effective load balancing for cluster-based servers employing job preemption. *Performance Evaluation*, 2008, 65(8): 606-622.

[19] Cormen T H, Leiserson C E, Rivest R L, Stein C. Introduction to Algorithms. Second Edition, MIT Press, 2001.

[20] PeKing University Application Server. http://forge.objectweb.org/projects/pkuas.

[21] Pacifici G, Segmuller W, Spreitzer M, Tantawi A. Dynamic estimation of CPU demand of Web traffic. In *Proc. the 1st International Conference on Performance Evaluation Methodologies and Tools* (*VALUETOOLS*), Pisa, Italy, Oct. 11-13, 2006, pp.26.

[22] Clark C, Fraser K, Hand S, Hansen J G, Jul E, Limpach C, Pratt I, Warfield A. Live migration of virtual machines. In *Proc. the 2nd ACM/USENIX Symp. Networked Systems Design and Implementation* (*NSDI*), Boston, USA, May 2-4, 2005, pp.273-286.

Yan Li received the B.Sc. degree in computer sciences from Beijing University of Technology in 2005. Currently, she is studying in the Institute of Software, School of Electronics Engineering and Computer Science, Peking University as a Ph.D. candidate. Her research interests include middleware, resource management and software engineering.

Feng-Hong Chen is an undergraduate in the Institute of Software, School of Electronics Engineering and Computer Science, Peking University. He has recommended for admission to Peking University for master degree. His research interest includes resource management for shared clusters.

Xi Sun was born in 1982. He received the Ph.D. degree in computer science from Peking University in 2009. His current research interests include software engineering, software component technology and intelligent software systems.

Ming-Hui Zhou was born in 1974. She received the Ph.D. degree in computer science from National University of Defense Technology in 2002. Now she is an associate professor of Institute of Software, School of Electronics Engineering and Computer Science, Peking University. She is a member of CCF. Her research interests include distributed computing, software measurement, etc.

Wen-Pin Jiao obtained his B.E. and M.E. degrees in computer software from East China University of Science and Technology in 1991 and 1997 respectively, and his Ph.D. degree in computer science from Chinese Academy of Sciences in 2000. After he finished his Ph.D. study, he went to University of Victoria, Canada as a post-doctoral fellow from 2000 to 2002. Now he is an associate professor in computer science and working with the School of Electronics Engineering and Computer Science at Peking University, China. He is a senior member of CCF. His research interests mainly include software engineering, intelligent software, multi-agent systems, and formal methods.

Dong-Gang Cao obtained his Ph.D. degree in computer science from Peking University in 2005, and then became a faculty member of the School of Electronics Engineering and Computer Science at Peking University, China. Now he is an associate professor in computer science. His research interests include software engineering, distributed systems, component-based software development.

Hong Mei is a professor at the School of Electronics Engineering and Computer Science, Peking University and the director of Special Interest Group of Software Engineering of CCF. He received his Ph.D. degree in computer science from Shanghai Jiaotong University in 1992. His current research interests include software engineering and software engineering environment, software reuse and software component technology, distributed object technology, software production technology and programming language.