Available online at www.sciencedirect.com
ScienceDirect
Future Generation Computer Systems 24 (2008) 631–643
FGCS FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/fgcs

RCT: A distributed tree for supporting efficient range and multi-attribute queries in grid computing

Hailong Sun[a,*], Jinpeng Huai[a], Yunhao Liu[b], Rajkumar Buyya[c]

[a] School of Computer Science and Engineering, Beihang University, Beijing, China
[b] School of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong
[c] Department of Computer Science and Software Engineering, The University of Melbourne, Australia

Received 10 April 2007; received in revised form 12 December 2007; accepted 23 December 2007
Available online 15 January 2008

Abstract

Resource discovery is of great importance in grid environments. Most of existing approaches treat all resources equally without any categorizing mechanism. We propose, Resource Category Tree (RCT), which organizes resources based on their characteristics represented by primary attributes (PA). RCT adopts a structure of distributed AVL tree, with each node representing a specific range of PA values. Though RCT adopts a hierarchical structure, it does not require nodes in higher levels maintain more information than those in lower levels, which makes RCT highly scalable. RCT is featured by self-organization, load-aware self-adaptation and fault tolerance. Based on RCT, commonly used queries, such as range queries and multi-attribute queries, are well supported. We conduct performance evaluations through comprehensive simulations.

Keywords: Resource discovery; Range query; Multi-attribute query; Grid computing

1. Introduction

Grid computing [9] aims at wide-area resource sharing and coordinated problem solving. Resource discovery is of paramount importance for achieving this goal. Grid information service (GIS) [17] is proposed to address resource discovery. Due to the large-scale, highly distributed and heterogeneous natures of grid environments, GIS faces many challenges. During the past years, several popular GIS systems have been designed, such as Globus MDS [7] and Condor gang-matchmaking [18]. P2P [13] and semantic-based [11,23] search are also introduced to deal with resource discovery.

In this study, we consider two challenges faced by GIS, i.e. efficiency and complex queries including range and multi-attribute queries. First, as grid systems usually involve millions of resources including computing power, storage, devices, data and so on, it is very important and challenging to obtain resource information in a short time so as to efficiently processing user jobs. Most of the existing GIS systems like MDS [7] simply organize resources based on some overlay, while information about all resources is treated equally without any categorizing mechanisms. This leads to traversing the whole overlay in order to search for desired resources. If resources can be categorized by some characteristics, we can further organize resources with similar characteristics by particular overlay. By this means, queries can be processed by searching a subset out of a large number of accessible resources. This is believed to be able to reduce the overhead greatly and improve resource discovery efficiency. Second, in order to schedule user jobs to the most appropriate resources, complex query mechanisms are required. Grid resources are usually characterized by sets of attributes, and users query resources by specifying the values of some resource attributes. A simple example is like "available memory = 500 MB", however more complex queries than that are desirable in most cases. Among of them are range and multi-attribute queries like "OS = linux and available memory >= 500 MB". There has been a lot of research work [3–5,10,14,25] on range and multi-attribute queries. Most of the work [3,5,10,25] assumes a DHT (Distributed Hash Table)-based infrastructure,

* Corresponding author. Tel.: +86 10 8233 9063; fax: +86 10 8231 6796.
E-mail addresses: sunhl@act.buaa.edu.cn (H. Sun), huai@buaa.edu.cn (J. Huai), liu@cs.ust.hk (Y. Liu), raj@csse.unimelb.edu.au (R. Buyya).

0167-739X/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2007.12.002

however DHT itself destroys data locality, which increases the overhead to process a range query. Mercury [4] supports multi-attribute range queries by creating routing hubs and organizing routing hubs into a circular overlay of nodes. Li et al. propose DPTree [14] to support various types of queries on multi-dimensional data in P2P systems based on balanced tree indexes.

In service grids that are widely accepted by research and industry communities, grid resources can be generally categorized into computational resources and grid services of system and application levels. Computational resources provide runtime support for services in both system and application levels. By computational resources, we mean hardware resources with capacity of supporting computation, such as clusters, storage, super computers and PCs. On the other hand, grid services provide certain processing functions like traditional software. And in our earlier work [12,21] we have presented the advantages of separating underlying computational resources from services by hot service deployment in a service grid [8]. With this separation, underlying resources and services from different providers are discovered and provisioned dynamically to meet specific nonfunctional requirements, such as low costs or high resource usage. This work is devoted to addressing the efficient resource discovery in service grids. In the rest of the paper, we will take the computational resource discovery as an example to introduce our proposed solution, and explain how the solution can be applied to service discovery.

In CROWN project [1], we developed many applications based on CROWN middleware. One of them is developed to factorize huge integers, which can be categorized as a computing intensive, one because powerful CPU resources are urgently required while storage requirement is trivial. Another one is DSS (Digital Sky Survey) [2] that retrieves data from a space telescope and provides a GUI interface for end users to query the star graph of a specified region. DSS needs at least 60 GB storage to store the data from the space telescope while processing of user queries does not require powerful computing power. With this experience, we observe that applications can be characterized by their requirements for computational resources. Intuitively the resource discovery efficiency will be improved if we can organize resources according to the characteristics of application resource requirements. For example, resources with powerful CPU capacity are organized as a category to serve computing intensive applications. This is how our idea is motivated.

Resources are usually described by a set of attribute-value pairs. Among all attributes of a resource, we choose one or several attributes that can best characterize the resource capacity of meeting application resource requirements as primary attributes (PA). We propose an overlay, RCT (Resource Category Tree), to organize computational resources based on PAs. With RCT, data locality is well preserved, which makes it possible to support efficient range queries.

Our major contributions are as follows:

- We identify the need to organize resources using categorizing mechanism so as to improve resource discovery efficiency and we propose an effective overlay, RCT, to organize resources in a self-organizing, self-adaptive and fault-tolerant manner.
- We propose load-aware self-adaptation algorithms, through which RCT nodes can achieve autonomic-access load balancing:
- Basing on RCT overlay, we propose algorithms to support four commonly-used resource queries and provide corresponding complexity analysis.
- We evaluate the performance of RCT through comprehensive simulations.

The rest of this paper is organized as follows. Section 2 presents related work. We give an overview of RCT definition and resource organization based on RCT in Section 3. Section 4 presents the design details. We describe the evaluation methodology and results in Section 5. Section 6 presents how RCT can be applied to service discovery, and we conclude this paper in Section 7.

2. Related work

Resource discovery is an important issue in grid environments. Two protocols (GRIP and GRRP) and two components (GIIS and GRIS) are proposed in Globus MDS [7] to construct a hierarchical grid information service. Condor [18] leverages ClassAd language to describe both queries and resources, and gang-matchmaking is proposed to match user queries with appropriate resources. In [24], a thorough performance evaluation of MDS and Condor is provided. P2P search technologies have also been adopted to address resource discovery in grids [13,15,22]. The above-mentioned approaches mainly focus on how to route user requests to target nodes, and the characteristics of application resource requirement are not considered. An overlay SOG is proposed in [16] to organize resources based on similarities of specific resource characteristics, using a hybrid P2P structure. A group is formed by a collection of nodes with some similarities in their characteristics and a leader is elected through gossip protocol. A group in SOG is similar to an RCT, but they are different in that resources in an RCT are further organized according to the value of primary attributes. Additionally, RCT considers application resource requirement when defining an RCT.

Through mapping resource key to resource locations, DHT (Distributed Hash Table) technologies, such as Chord [20] and CAN [19], can effectively address target resource by searching a limited number of nodes. But to support range and multi-attribute queries, additional efforts are needed. Many resource-discovery mechanisms based on DHT [3,5,6,10,25] have been proposed over the past few years to address range and/or multi-attribute queries. For example, the study in [6] presents a DHT-based peer-to-peer approach for computational-resource discovery. The static and dynamic parts of resource attributes are combined into a Resource ID that serves as a key in a Pastry-based system. The resources are represented as overlapping arcs on a Pastry ring. The beginning of an arc represents the static attribute set and the length represents the spectrum of dynamic states. However, no mechanisms of load-aware adaptation are

provided to eliminate possible bottlenecks caused by hot spot query. The authors in [10] propose a logical binary tree RST (Range Search Tree) on the basis of DHT infrastructure to support range queries. One advantage of RST is that it does not need to maintain the tree structure dynamically because the domain of an attribute is split into 2^n sub ranges beforehand and each node can deduct the tree structure locally. The dynamic RST is adaptive to the query and registration load. However, RST requires resources to register with many nodes whose responsible ranges cover the resource attribute value, incurring huge overhead in dynamic grid systems where resources need to frequently update status. In practice, as DHT itself destroys data locality due to the use of randomizing hash functions, it will generate many *lookup (key)* operations to process a range query in DHT-based solutions, which is believed to be increase much overhead compared to data locality-preserving solutions.

There also has been research work [4,14] on range and multi-attribute query that is not based on DHT. Mercury [4] supports multi-attribute range queries by creating routing hubs and organizing routing hubs into a circular overlay of nodes. A routing hub is a logical collection of node sin the system and responsible for a specific attribute in the overall schema. When there are a large number of attributes, the overlay maintenance overhead is very large. Different from Mercury, RCT organizes resources on the basis of selected attributes (i.e. primary attributes), not every attribute. Li et al. propose DPTree [14] to support various types of queries on multi-dimensional data in P2P systems based on balanced tree indexes (R-Tree). DPTree adopts a skip graph-based overlay and maps a logical R-Tree to the overlay network in a distributed manner. RCT adopts a balanced binary search tree structure and corresponding overlay network, and RCT provides a load-aware self-adaptation mechanism.

3. RCT overview

In this section, we describe RCT definition and the architecture of resource organization based on RCT.

3.1. Resource description

The attribute-based approach is widely adopted for describing resources in grid-computing environments. In this paper, we also choose this approach to describe computational resources. Each computational resource is characterized by a set of attribute-value pairs. In practice, we are mainly concerned about dynamic attributes (e.g. CPU load) of a resource in a real computing environment because dynamic status represents available capacity of a resource.

3.2. RCT-resource category tree

Grid applications can be characterized by their requirements for computational resources, e.g. computing intensive and data-intensive applications. In turn, we can categorize computational resources based on certain resource characteristics that can meet application resource requirements. By doing so, resource discovery is performed on specific resource categories efficiently. For example, we know that resources with huge storage can better serve a data-intensive application, thus we can organize them together based on an overlay structure.

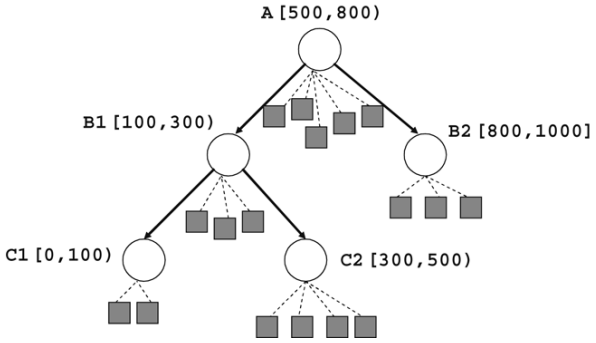

Fig. 1. An example of RCT.

Furthermore, we observe that values of most resource attributes are numerical, e.g. values of disk size. And attributes whose values are not numerical can be converted to be numerical through certain mathematical methods. Based on this consideration, RCT adopts an AVL tree (or *balanced binary search tree*) overlay structure to organize resources with similar characteristics. The attribute that can best describe the characteristic of resources organized by an RCT is named a *primary attribute* or *PA*. Fig. 1 is an example of RCT. The chosen PA is *available memory size*, and the value domain of available memory ranges from 0 to 1000 MB.

Compared to traditional AVL, each node of RCT manages a range of values, instead of a single value. Each node only needs to maintain its connection with direct child nodes and parent, and operations like registration, updating and query can start from any node. Unlike in traditional AVL structure, higher-level nodes of RCT are not required to maintain more information or bear more load than those in lower levels, which provides the basis for RCT to scale easily.

Suppose D is the value domain of the PA of an RCT. Each node n of an RCT is responsible for a subrange of D, or D_n. All resources with PA values belonging to D_n register themselves to node n. We name each RCT node an HR (Head of a subrange). And terms of "HR n" and "node n" will be used interchangeably in the rest of this paper. In Fig. 1, the circles denote HRs, while the squares below an HR denote computational resources registered with an HR.

Suppose N is the total number of HRs in an RCT, *lc(n)* and *rc(n)* are the left and right child nodes of HR n respectively. Since an RCT is a binary search tree, we have the following observations:

$$\bigcup_{i=1}^{N} D_i = D \tag{1}$$

$$D_i \cap D_j = \phi, \quad \forall i, j \in [1, N] \tag{2}$$

$$D_{lc(i)} < D_i < D_{rc(i)}, \quad \forall i \in [1, N]. \tag{3}$$

We say $D_i < D_j$ if the upper bound of D_i is less than the lower bound of D_j, e.g. $[1, 2] < [3, 4]$.

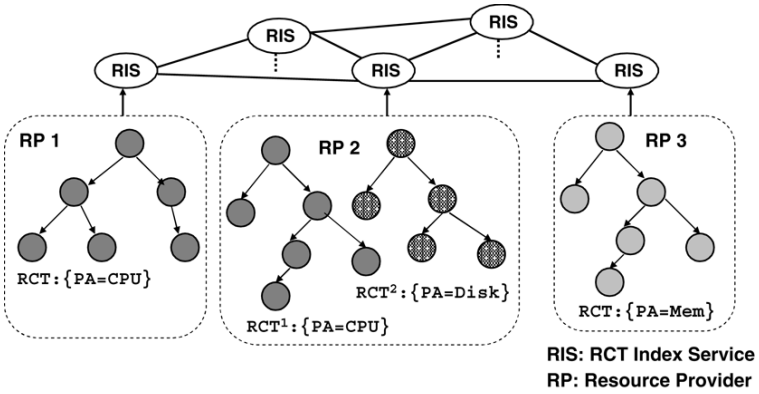

Fig. 2. Resource organization with RCT.

If the ranges of node i and node j, i.e. D_i and D_j, are adjacent, node i is referred to as a neighbour of node j, and vice versa. If node i is a neighbor of node j and $D_i < D_j$, node i is called left neighbour of node j (denoted by *L-neighbor(j)*); node j is called right neighbour of node i (denoted by *R-neighbour(i)*). Note there are two exceptions: the leftmost HR and the rightmost HR, the former has no left neighbor and the latter has no right neighbour.

As shown in Fig. 1, C2 and B2 are neighbours of A, while C2 is L-neighbour of A and B2 is R-neighbour of A. Note that C1 does not have a left neighbour and B2 has no right neighbour.

3.3. Organizing resources with RCT

As resources are owned and managed by different resource providers, providers may define different PAs for their resources, which results in constructing multiple RCTs. In Fig. 2, we present a 2-layer architecture for organizing resources across resource providers by using RCT. In the lower layer, each resource provider defines a set of PAs that can best describe their resources. Based on PAs, resources are organized through a certain number of RCTs. To enable wide area resource discovery across different providers, an RCT index service (RIS) is deployed by each service provider in the upper layer. RIS is a basic service that stores information about PAs of a provider and entry points of RCTs. RISs can be implemented, e.g. as web services or grid services, and find each other using services like UDDI.

In practice, a resource may have many attributes, but only a few of them are chosen as primary attributes. So there will not be too many RCTs. When a query request cannot be satisfied by a resource provider, the RIS will contact other RISs to recommend another resource provider for further discovery operations.

4. Design of RCT

Three goals are considered in RCT design: (1) *self-organization*, which is very important for dynamic grid environment. Manual operations should be as few as possible so as to allow resources to join or leave freely and keep the system scale easily. (2) *load-aware self-adaptation*, which is necessary to improve the availability and scalability when load of registration, query and updating is not balanced among HR nodes. (3) *fault-tolerance*, which is critical to handle unexpected failures of RCT nodes. In this section, we present how the design of RCT achieves these three goals. Following that, we describe the resource searching algorithm based on RCT.

4.1. Bootstrapping an RCT

As an RCT consists of a set of HRs, before building an RCT we need to consider how an HR is set up. To make RCT a self-organizing system, we require HRs to be chosen automatically from resources themselves.

At first thought, it is a simple way to choose an HR randomly from resources. Nevertheless, if the chosen HRs are unstable or weak in capacity, the instability will affect the availability of RCT, and the weak capacity will lead to bottlenecks. Hence, we need to consider both *availability* and *capacity* when choosing a qualified HR. A resource with long online time means that it is much more stable and available. Additionally, a resource with powerful capacity can ensure that it is capable of serving as an HR to manage a set of resources. The availability of a resource is defined as its online probability $p(p = t_{\text{on}}/(t_{\text{on}} + t_{\text{off}}), t_{\text{on}})$ and t_{off} are the online and offline time during a past period of time respectively); the capacity is measured by the computing power c (e.g. $c =$ CPU Frequency).

At the beginning of the initialization stage, no RCTs or HRs exist in a grid environment. An RCT index service (RIS) is configured and deployed to maintain the RCT information including primary attributes, their value domains, and entry points of at least one HR. The procedure of building an RCT is as follows:

(1) A resource queries RIS to get information about RCT configurations of PAs.
(2) The resource checks whether it can satisfy the condition of being an HR of an RCT. If yes, go to step (3); else, the resource is not qualified to be organized.
(3) The resource sends an HR-application request to RIS. The request contains information about which RCT it aims to build and data of its *availability* and *capacity*.

(4) The *RIS* stores the data of the candidate resources. When the candidates for an RCT reach a certain number, RIS will compare the data of *availability* and *capacity* to select the most qualified one as the first HR of the relevant RCT. Then the RIS notifies other candidate resources to register with the selected HR.

The RIS serves as the bootstrapping service for building an RCT. After an RCT is built, resources can get an entry point to the RCT from RIS, but this is not the only approach. The information can be cached locally for later use, and a resource may get this information from other resources. Note that RIS is only responsible for building the first HR of an RCT, and the building of other HRs leaves to the RCT itself, which will be introduced in next section.

4.2. Load-aware self-adaptation

Resource maintenance. Due to the dynamic nature of resources, we need to address resource joining, leaving and status changing.

When a computational resource connects to a grid system, it will first try to find an RCT to register itself. This can be done through RCT index service. Eventually a resource will know the address of at least one HR, then sends its joining request to the HR. According to the PA value of the resource, the HR checks if it should manage the incoming resource. If yes, the resource is registered with the HR; otherwise, the HR will traverse RCT to find the HR that the resource should register with. After registration, a resource will periodically update its status including PA value to the corresponding HR. However, if the PA value of an incoming resource lies out of the ranges of all HRs in an RCT, this means the current RCT cannot accept the registration of the resource. Then the incoming resource can find another appropriate RCT to register with.

A resource can leave without any notification. An HR should recognize the leaving of a resource as early as possible. For this purpose, a resource is required to send updating message to its HR periodically even its status is unchanged. The updating message is empty in case of unchanged resource status. On the other side, an HR will remove the related resource information if it has not received updating message from a resource for a period of time.

An HR only manages resources whose PA values belong to the HR's responsible range. When the PA value of a resource is out of an HR's range due to dynamic changes, the HR will traverse RCT to find a proper HR to transfer the resource to. For the transferred resources, this is similar to a re-registration to a new HR.

Note that the resource status of an HR itself also changes dynamically. One interesting question: when an HR's PA value no longer belongs to the subrange it is responsible for, will the HR be degraded to be a common computational resource? The answer is no, otherwise it will cause RCT to be unstable. A resource that serves as an HR also serves as a common resource as well. Therefore, as a resource, an HR is also managed by another HR that is not necessary to be itself.

Algorithm 1: Balance an overloaded HR n

```
1:  //Initialize splitting policy
2:  policy = AVS;
3:  //Retrieve neighbors' load information
4:  load_n1 = getLoad (L-neighbor (n));
5:  load_n2 = getLoad (R-neighbour (n));
6:  if load_n1 <= l_warning AND load_n2 <= l_warning then
7:    // split D_n into 3 subranges
8:    split (D_n, D_1, D_2, policy);
9:    // transfer resources in D_1 from n to n1
10:   transferLoad (n, n1, D_1);
11:   //transfer resources in D_2 from n to n2
12:   transferLoad (n, n2, D_2);
13: else if load_n1 <= l_warning then
14:   // split D_n into 2 subranges
15:   split (D_n, D', policy);
16:   // transfer resources in D' from n to n1
17:   transferLoad (n, n1, D');
18: else if load_n2 <= l_warning then
19:   // split D_n into 2 subranges
20:   split (D_n, D', policy);
21:   // transfer resources in D' from n to n2
22:   transferLoad (n, n2, D');
23: else
24:   split (D_n, D', policy);
25:   select an HR n' from resources in D';
26:   Insert HR n' into current RCT;
27:   balanceTree();
28: end if
```

Load-aware adaptation. There are several cases in which an HR can be overloaded. For example, if an HR is responsible for a big range of PA values or the range of an HR is a "hot spot", a large number of messages of registration, updating and query will overwhelm the relevant HR. Therefore if no appropriate measures are taken, the overloaded HRs will become the bottlenecks. On the one hand, when an HR is overloaded, its subrange should be split, which results in new HRs being chosen or some resources are transferred to others. On the other hand, when an HR is light loaded and no other HRs transfer resources to it as well, we should consider deleting it and merging its range with other HRs responsible for adjacent ranges. Deleting a light-loaded HR and merging corresponding subranges can reduce the average search length. In all, RCT must have the ability to adapt according to load state.

Before going further, a metric should be defined to represent an HR's load l. As the data for describing a resource is only a few attribute-value pairs, managing resources will not consume much storage of an HR. We use CPU load to represent an HR's load. Light-loaded threshold l_{light} and overloaded threshold l_{over} are defined respectively. Additionally, a warning threshold l_{warning} is defined ($l_{\text{light}} < l_{\text{warning}} < l_{\text{over}}$) to avoid oscillation problems, i.e. a node easily may become overloaded after receiving load from others. When the load of an HR is above $l_{\text{warning}}$and below l_{over}, it indicates that the HR is near to be overloaded and cannot accept new load

transferred from other HRs. For example, when l is greater than 90% ($l_{\text{over}} = 90\%$), an HR is considered as overloaded; when l is less than 10% ($l_{\text{light}} = 10\%$), an HR is light loaded when l is less than 80% ($l_{\text{warning}} = 80\%$) and greater than 10%; an HR is ready to help its neighbours to balance load.

Note that an HR is also a computational resource that is used to process user jobs. To ensure the user job processing will not affect an HR's organizing resources in RCT, a resource reservation mechanism is needed. For example, if a resource acts as an HR, 20% of CPU time will be reserved for its responsibility in RCT. This will rely on resource manager's reservation functionality. Hence the definitions of l_{light}, l_{over} and l_{warning} are based on the reserved capacity for organizing resources.

Suppose D_n is the range an HR n is responsible for and HR n becomes overloaded. According to the above analysis, D_n will be split to balance the load. We design two policies for splitting a range: *Average Split* (AS) and *Analysis of Variance-based Split* (AVS). With AS, D_n is split evenly into two or three subranges; AVS considers load distribution across D_n based on analysis of variance. For example, suppose [10,100] is a range, and load is mainly distributed in [80,100]. With AS, the range may be split into [10,55) and [55,100]; while with AVS, the splitting results can be [10,90) and [90,100]. In order to be consistent with the three observations presented in Section 3.2, an HR only transfers load to two neighbours, which means D_n is split at most into three subranges for each time.

Algorithm 2: Resign a light-loaded HR n

```
1:  //Initialize splitting policy
2:  policy = AS;
3:  //Retrieve neighbours' load information
4:  load_n1 = getLoad (L-neighbour (n));
5:  load_n2 = getLoad (R-neighbour (n));
6:  while (load_n1 > l_warning AND load_n2 > l_warning) do
7:     sleep (t);
8:  if load_n1 <= l_warning AND load_n2 <= l_warning then
9:     // split D_n into 2 sub ranges: D_1 < D_2
10:    split(D_n, D_1, policy);
11:    D_2 = D_1; D_1 = D_n;
12:    // transfer resources in D_1 from n to n1
13:    transferLoad (n, n1, D_1);
14:    //transfer resources in D_2 from n to n2
15:    transferLoad (n, n2, D_2);
16:  else if load_n1 <= l_warning then
17:    D_1 = D_n;
18:    // transfer resources in D_n from n to n1
19:    transferLoad (n, n1, D_1);
20:  else if load_n2 <= l_warning then
21:    D_2 = D_n;
22:    // transfer resources in D_n from n to n2
23:    transferLoad (n, n2, D_2);
24:  end if
25:  resign (n);
26:  balanceTree();
```

Algorithm 3: Complete load-aware adaptation for HR n

```
1:  while (true) do
2:  //Retrieve HR n's load information
3:  load_n = getLoad (n);
4:  if load_n >= l_over then
5:     balance the load of HR n; //Algorithm 1
6:  else if load_n <= l_light then
7:     resign HR n; //Algorithm 2
8:  end if
9:  sleep (t);
```

Algorithm 1 is designed to balance an overloaded HR. We first initialize the splitting policy as AVS (Line 2); then the load of two neighbours is obtained (Lines 4–5); according to the load status of neighbours, the range is split and load is transferred to relevant neighbours (Lines 6–22); in case both neighbours are not available, a new HR is selected to balance the load, and RCT itself needs to be balanced (Lines 24–27).

In contrast to the overloaded case, when HR n is light loaded, it will try to merge itself with its neighbours and resign the HR post so as to reduce the average search length. As the average search length of RCT has great impact on searching efficiency, the merge of light-loaded HRs will lead to higher efficiency of resource discovery. This procedure is shown in Algorithm 2.

Here the splitting policy is set as AS (Line 2), because the whole range of current HR has light load; then the neighbours' load is obtained (Lines 4–5); if both of neighbours' load are above warning threshold, it has to wait for next period to try again (Lines 6–7); the load is transferred to left neighbour and/or right neighbour (Lines 8–24); eventually the HR resigns and operation is performed to balance RCT (Line 25–26).

Based on Algorithms 1 and 2, we have the complete load-aware adaptation algorithm, as shown in Algorithm 3. The algorithm runs periodically at each HR to ensure the HR works in a normal load state.

HR failures. The leaving of an HR can be categorized as normal leaving and abrupt failures without notification.

If an HR leaves normally, it will choose a new HR from resources it manages to replace itself. However, the unexpected failure of an HR is much more complicated, which makes the child trees of the failed HR disconnected with the other HRs of the relevant RCT. Therefore, it is desirable to have a fault-tolerant design to handle such failures. In practice, the HR failures can be caused by software or hardware breakdown (e.g. memory overflow), or network disconnection.

We detect an HR's failure by sending keep-alive messages periodically between a node and its parent. As a result, the parent node or direct child nodes of a failed HR will first notice the failure of an HR. One straightforward way of recovering RCT is to locate the disconnected child trees and assign new parent HRs for them. This procedure is much more complex if several HRs fail simultaneously. In that case, an HR is required to maintain information of either all of predecessors or offspring, which means an HR's leaving or joining has to be notified to many other HRs. In case of frequent HRs joining or leaving, this will definitely add a large overhead to the whole system.

Instead we choose a rather simple but effective approach that is based on redundancy. Note that the online time is an important factor in defining the criteria for selecting an HR. Therefore, an HR has better availability than other resources. And the probability of the simultaneous failure of multiple HRs will be small. Based on the above analysis, we require each HR to have an alternate backup. When the primary HR crashes, the backup HR will be activated to work as primary HR and a new backup is selected at the same time. Even in case that both HRs fail, the disconnected resources can join the RCT again later after they realize this.

4.3. Searching with RCT

The ultimate goal of RCT design is to process resource queries and return resources that best meet users' query requirements. Given the resource organization defined by RCT, we present the search mechanisms in this section.

First, we identify four types of commonly used queries from two dimensions, as shown in Fig. 3. Here we provide examples for these four types of queries: Q1: "Available storage = 90 GB"; Q2: "CPU load $<$ 50%"; Q3: "Available storage = 100 GB AND OS = WindowsXP"; Q4: "Availablememory $>$ 256 MB AND CPU load $<$ 80%". We refer to searches corresponding to the four types of queries as Q1-search, Q2-search, Q3-search and Q4-search respectively.

RCT supports all these four types of queries while we only present searching algorithm for Q2-search and give description to show how Q2-search can be extended easily to support the other three. In order to balance the query load, the algorithm is designed to allow a query starting from any node of an RCT, instead of only from the root node.

The Algorithm 4 is designed for Q2-search. D_q is supposed to be the range of a user query for a primary attribute (PA), and D_n is the range that HR n is responsible for. $D_{l-\text{child}T(n)}$ and $D_{r-\text{child}T(n)}$ are ranges that the left child tree and right child tree of n are responsible for respectively. The algorithm is designed as a recursive function. The query results will be aggregated in a data structure called *ResultSet*. We first split D_q into three subranges: D_0, D_1 and D_2 (Line 4). D_0 is the intersection of D_n and D_q; D_1 and D_2 are subranges of D_q that are adjacent to the left and right ends of D_0 respectively. If D_0 is not empty, search is performed locally for the range of D_0 (Line 5); if D_1 is not empty, search is performed in the left child tree (Line 6); if D_2 is not empty, search is performed in the right child tree and/or at parent node depending on the relationship of D_2 and $D_{r-\text{child}T(n)}$ (Lines 7–9); Finally, the complete result set containing the results from all the search processes is returned (Lines 10).

To implement the above algorithm, each HR n is required to know the total range that its child tree is responsible for. With this information, $D_{r-\text{child}T(n)}$ and $D_{l-\text{child}T(n)}$ can be calculated. To achieve this, when load is transferred to/from a node, the node will not only update range information of its child trees but also notify relevant child nodes to update range information of their respective child trees.

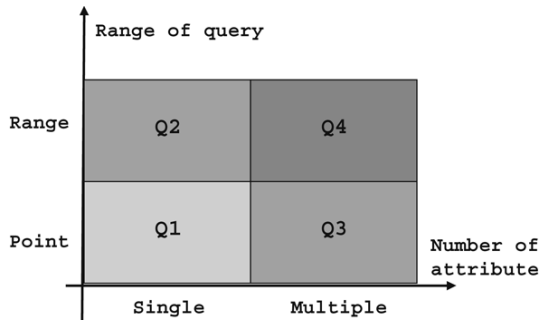

Fig. 3. Four types of commonly used queries.

With the Q2-search algorithm, Q1-search is much simpler because Q1-search is indeed a Q2-search with the D_q of only one element.

Q4-search is the most complicated among the four types of searches. Based on Q2 search, it can be dealt with in the following three steps. (1) We first identify a PA in query constraints. (2) Then based on Algorithm 4, searching is performed with the query range of selected PA. (3) If a node satisfies the constraints on selected PA, further search will be performed at current node on constraints of other attributes. For example, if a query is "Available storage $>$70 G AND CPU Load $<$ 20%" and "Available storage" is identified as PA, search will be performed on an RCT whose PA is "Available storage". If HR n satisfies the constraint of "Available storage $>$ 70G", the resources managed by HR n will be searched further, based on "CPU Load $<$ 20%".

And Q3-search, which is multi-attribute and point search, is a simplified version of Q4-search.

As a query can start from any node, the search length of a point query will be $2 \log N$ at most. For the range query, the complexity is highly dependent on the length of a query range, and we will study this in Section 5.

Algorithm 4: Q2-search from HR n

```
1:  func search (D_q, n) : ResultSet
2:      ResultSet rs_1, rs_2, rs_3, rs_4;
3:      D_0 = D_q ∩ D_n;
4:      split D_q − D_0 into D_1 and D_2 (D_1 < D_2);
5:      if D_0! = φ then   rs_1 = perform local search for D_0;
6:      if D_1! = φ then   rs_2 = search (D', lc(n));
7:      if D_2! = φ then   D' = D_2 ∩ D_r-child_T(n);
8:      if D'! = φ then    rs_3 = search (D', rc(n));
9:      if D_2 − D'! = φ AND parent(n)! = null then
        rs_4 = search(D_2 − D', parent(n));
10:     return rs_1 ∪ rs_2 ∪ rs_3 ∪ rs_4
11: end func
```

One thing to note is that an incentive mechanism is introduced to the HR's local search. Considering that HRs contribute their capacity for RCT, we increase their priority to be searched. That means if two resources including an HR meet user query requirements and the user only need a limited number of resources, the HR will be returned to a user with a higher priority.

4.4. Parallel search

AS each HR is responsible for a different range of value domain of a PA, the bigger length of a query range means more hops are needed for processing a query. Thus Algorithm 4 could be inefficient if the length of a query range is very large. If the ranges of all HRs are known beforehand, we can split the query range into multiple subranges so that each subrange matches an HR exactly. Thus searches can be performed in parallel for each subrange. In Section 4.2, we mention that in order to maintain parent-child relationship of HRs, a child is required to send keep-alive message to its parent periodically. An HR can utilize this message to piggyback information about the HRs and their responsible ranges. In this way, each HR will know others' ranges, and parallel search can be implemented. When the range of an HR changes due to self-adaptation, the change will be known to all HRs after at most $\log_2 N$ keep-alive periods. Before receiving notification about range changes, using stale information can cause some parallel searches to fail. In such a case, Algorithm 4 will be used for searching.

4.5. Complexity analysis

In this section, we analyse the complexity of searching algorithms with RCT. From Section 4.3, we can see that range search and multi-attribute search are just variations of Q2-search. And Q2-search can be reduced to Q1-search if parallel search introduced in Section 4.4 is adopted. Thus we only analyse the complexity of Q1-seach that is a random search against RCT.

We start from a complete BST (Binary Search Tree) with n nodes and the height of h.

Suppose the entrance of searching is $node_k$ ($k > 2$) that lies in the kth level of the BST, then the target node $node_{target}$ is possible to locate in the 3 shaded areas shown in Fig. 4. We use TSL(M, subtree(N)) to denote the total search length when the searching starts from node M and traverse all the nodes in the subtree rooted at node N, and ASL (Average Search Length) to denote the average number of nodes that are traversed when the target node is randomly chosen with equal probability.

Case 1:

For all $node_{target}$ in the subtree rooted at $node_k$, i.e. $node_{target} \in subtree\ (node_k)$, the depth of the subtree is $h - k + 1$. So the TSL from $node_k$ to all the nodes within the subtree is

$$\begin{aligned}&\mathrm{TSL}(node_k, subtree(node_k))\\ &= (h - k + 1 - 1) \times 2^{h-k+1} + 1\\ &= (h - k) \times 2^{h-k+1} + 1.\end{aligned}$$

Case 2:

For all $node_{target}$ on the path from $node_k$ to root, i.e.$node_{target} \in path(node_k, root)$, given that there are $k - 1$ nodes on the path, the TSL from $node_k$to all the nodes on the path is

$$\begin{aligned}&\mathrm{TSL}(node_k, path(node_k, root)) = 2 + 3 + 4 + \cdots + k\\ &= \sum_{i=1}^{k-1}(i + 1) = \frac{(k + 2)(k - 1)}{2}.\end{aligned}$$

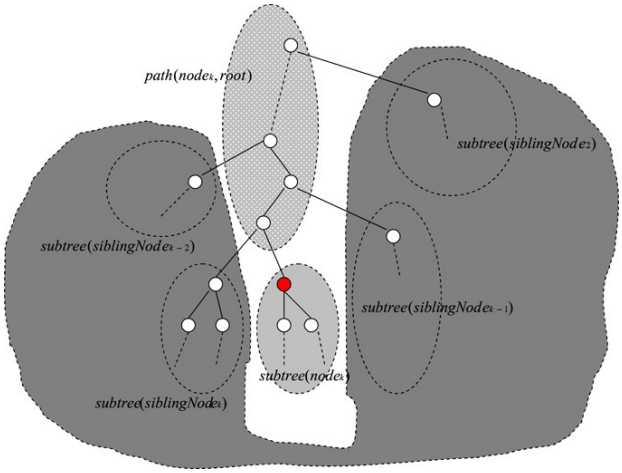

Fig. 4. Searching from a node in the kth level.

Case 3:

For all $node_{target}$ in the subtree with a $(k - 1)$th level node on $path(node_k, root)$ as its parent, i.e. $node_{target} \in subtree(siblingNode_k)$, the TSL from $node_k$ to all the nodes within the subtree is

$$\begin{aligned}&\mathrm{TSL}(node_k, subtree(siblingNode_k))\\ &= 2 \times (2^{h-k+1} - 1) + (h - k) \times 2^{h-k+1} + 1.\end{aligned}$$

Similarly, for all $node_{target}$ in the subtree with a $k - 1$ level node on $path(node_k, root)$ as its parent, i.e. $node_{target} \in subtree(siblingNode_{k-1})$, the TSL from $node_k$ to all the nodes within the subtree is

$$\begin{aligned}&\mathrm{TSL}(node_k, subtree(siblingNode_{k-1}))\\ &= 3 \times (2^{h-k+2} - 1) + (h - k + 1) \times 2^{h-k+2} + 1.\end{aligned}$$

It can be seen that if $node_{target}$ is in the subtree with a $(k-i)$th level node on $path(node_k, root)$ as its parent, i.e. $node_{target} \in subtree(siblingNode_{k-i+1})$, the TSL from $node_k$ to all the nodes within the subtree is

$$\begin{aligned}&\mathrm{TSL}(node_k, subtree(siblingNode_{k-i+1}))\\ &= (i + 1) \times (2^{h-k+i} - 1) + (h - k + i - 1) \times 2^{h-k+i} + 1\\ &= (h - k + 2i) \times 2^{h-k+i} - i - 1.\end{aligned}$$

Thus, the TSL from $node_k$ to all the nodes within the neighbouring subtrees is

$$\begin{aligned}&\mathrm{TSL}(node_k, siblingSubtree(node_k))\\ &= \sum_{i=1}^{k-1} TSL(subtree(siblingNode_{k-i+1}))\\ &= \sum_{i=1}^{k-1}[(h - k + 2i) \times 2^{h-k+i} - i - 1]\\ &= (h - k) \times 2^{h-k} \times (2^k - 2) + 2^{h-k+1}\\ &\quad \times [(k - 2) \times 2^k + 2] - \frac{(k + 2)(k - 1)}{2}.\end{aligned}$$

Hence, from the results of Cases 1–3, we can obtain the TSL from $node_k$ to all the nodes within the BST as follows:

$$\begin{aligned}\mathrm{ASL}_{total} &= \frac{\mathrm{TSL}_{total}}{SearchTimes_{total}} \\ &= \frac{[2^h \times (h-4)+1] \times (2^h-1) + 2^h \times [(h-1) \times 2^h + 1] + h \times 2^{h+1}}{(2^h-1)^2} \\ &\approx h-4+h-1 = 2h-5\end{aligned}$$

Box I.

$$\begin{aligned}\mathrm{ASL}_{total} &= \sum_{k=1}^{h} [p(node_k) \times \mathrm{ASL}(node_k)] \\ &= \sum_{k=1}^{h} \left[\frac{2^k}{2^h-1} \times \frac{2^h \times (h+k-4) + 2^{h-k+2} + 1}{2^h-1}\right] \\ &\quad \times \frac{[2^h \times (h-4)+1] \times (2^h-1) + 2^h \times [(h-1) \times 2^h + 1] + h \times 2^{h+1}}{(2^h-1)^2} \\ &\approx h-4+h-1 = 2h-5\end{aligned}$$

Box II.

$$\begin{aligned}&\mathrm{TSL}(node_k, subtree(root)) \\ &\quad = \mathrm{TSL}(node_k, subtree(node_k)) \\ &\qquad + \mathrm{TSL}(node_k, path(node_k, root)) \\ &\qquad + \mathrm{TSL}(node_k, siblingSubtree(node_k)) \\ &\quad = (h-k) \times 2^{h-k+1} + 1 + \frac{(k+2)(k-1)}{2} \\ &\qquad + (h-k) \times 2^{h-k} \times (2^k-2) + 2^{h-k+1} \\ &\qquad \times [(k-2) \times 2^k + 2] - \frac{(k+2)(k-1)}{2} \\ &\quad = 2^h \times (h+k-4) + 2^{h-k+2} + 1.\end{aligned}$$

Then the total TSL from each node within the BST to all the nodes within the BST is:

$$\begin{aligned}&\mathrm{TSL}_{total} \\ &\quad = \sum_{k=1}^{h} [NumberOf(node_k) \times \mathrm{TSL}(node_k, subtree(root))] \\ &\quad = \sum_{k=1}^{h} \{2^{k-1} \times [2^h \times (h+k-4) + 2^{h-k+2} + 1]\} \\ &\quad = [2^h \times (h-4)+1] \times (2^h-1) + 2^h \\ &\qquad \times [(h-1) \times 2^h + 1] + h \times 2^{h+1}.\end{aligned}$$

With our three assumptions mentioned above, we have the average search length ASL: See Box I

The above result can also be obtained through the following approach, where $p\ (node_k)$ is the probability of the presence of all the kth-level nodes in a BST:

$$\begin{aligned}\mathrm{ASL}(node_k) &= \frac{\mathrm{TSL}(node_k, subtree(root))}{2^h-1} \\ &= \frac{2^h \times (h+k-4) + 2^{h-k+2} + 1}{2^h-1}.\end{aligned}$$

Thus, we have the following result in Box II:

It can be easily calculated that when $h <= 2$, the ASL is one. Thus we have the conclusion that the average length of random search with complete BST is roughly equal to $2h-5$. Here, we give an estimation of average length of random search with AVL.

The difference between a complete BST and an AVL is that the number of leaf nodes in an AVL can ranges from 1 to 2^{h-1}. Therefore, the average search length will be less than that of complete BST with the same depth. Furthermore, the average search length should be larger than that of complete BST with $h-1$ depth. Then, we can conclude

$$\begin{cases}\mathrm{ASL}_{\mathrm{AVL}} \in (2h-7, 2h-5], & h \geq 3 \\ \mathrm{ASL}_{\mathrm{AVL}} = 1, & h \leq 2.\end{cases} \tag{4}$$

For example, the average search length of an AVL with 100 nodes is larger than seven and less than/equal to nine.

5. Performance evaluation

5.1. Simulation methodology

The RCT algorithms are simulated through an event-driven approach with Java codes. The simulated RCT has 100 nodes (HRs). The value domain D of selected PA is [0,100], and D is evenly split across the HRs, which means the length of an HR's subrange is one. The query arrival is modelled as a Poisson distribution.

The metrics we use are as follows: (1) *Number of queries per node*. This metric is used to evaluate the average query load of an HR in different situations. With this metric, we can know the overhead of RCT when facing different numbers of simultaneous queries. (2) *Average Search Length (ASL)*. ASL indicates the average number of HRs that a request is passed before being processed. Combining metrics (1) *and* (2), we can evaluate the efficiency of resource discovery. (3) *Standard deviation of query load*. As RCT has the ability of load-aware

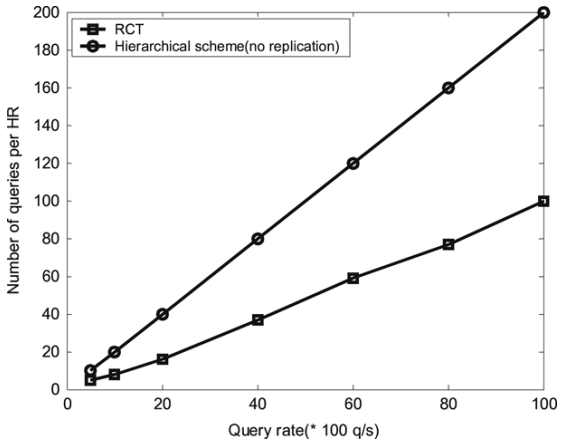

Fig. 5. Query rate vs. Query load per HR.

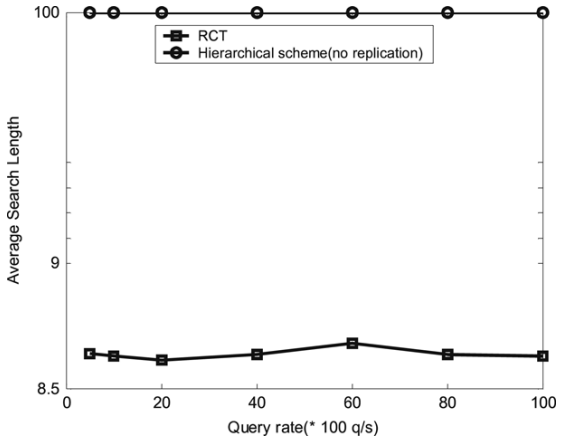

Fig. 6. Query rate vs. ASL.

adaptation, this metric is used to evaluate how a load is balanced across HRs. The smaller value of this metric indicates a better balancing effect.

5.2. Evaluation results

We design three experiments to evaluate RCT based on the three metrics defined above.

In first experiment, we evaluate RCT performance in terms of metric (1) and (2) through varying query rate. The query rate varies from 500 per second to 10,000 per second, and queries are distributed equally to 100 HRs. The queries in this experiment are Q1 queries that are single attribute and point queries. We compare RCT performance with hierarchical scheme. Globus MDS [7] is a hierarchical scheme, but it allows data replication across levels, and here we do not consider replication. From Fig. 4, we observe that RCT incurs less overhead to each node than hierarchical scheme. To show results better, the query load of an hierarchical scheme is reduced by 50 times in Fig. 5. Fig. 6 shows that a query is processed with smaller ASL than an hierarchical scheme under different query rates. From the conclusion drawn from Section 4.5, we know the ASL with RCT is between 8.5 and 9. We can say RCT outperforms an hierarchical scheme (no replication) in all cases.

As range query is widely used in grids, our second experiment is to evaluate how RCT performs with different lengths of query ranges. The lengths of query range vary from 1 to 100. Fig. 7 shows the results in terms of metric (1) using Algorithm 4. As the length of query range increases, the average load of an HR increases linearly. This is because query ranges are split into more subranges in the process of searching across RCT when the length of query range increases. Fig. 8 plots the comparison of parallel search with Algorithm 4 in terms of metric (2). The ASL with Algorithm 4 and parallel search increases linearly along with the length of query range, although parallel search can reduce ASL to some extent. Indeed, parallel search can reduce the total search time greatly by performing subsearches simultaneously. Note that although the second experiment is designed for single attribute and range queries, the results are similar with multi-attribute and range queries. This is because multi-attribute queries are first processed by performing search against the PA attribute contained in the given query constraints, and the constraints for other attributes will be processed locally in an HR node.

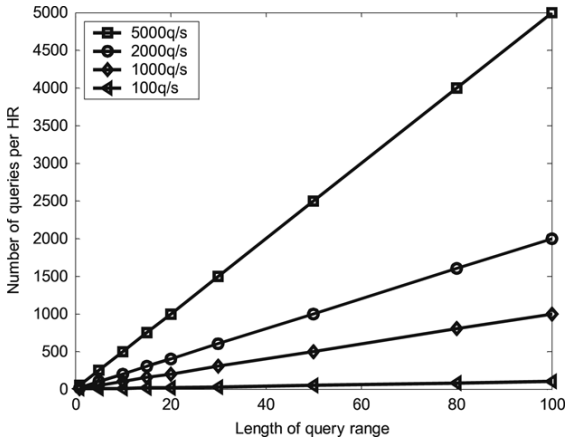

Fig. 7. Length of query range vs. Query load per HR.

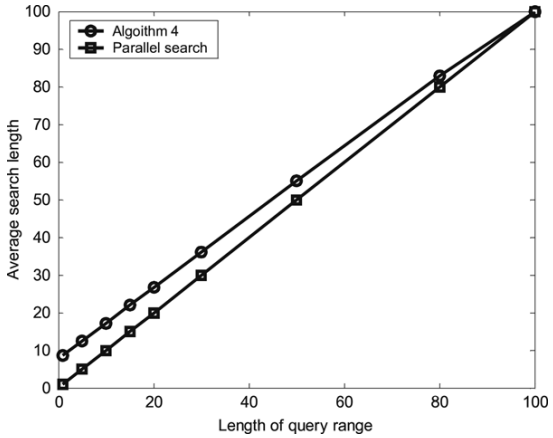

Fig. 8. Length of query range vs. ASL.

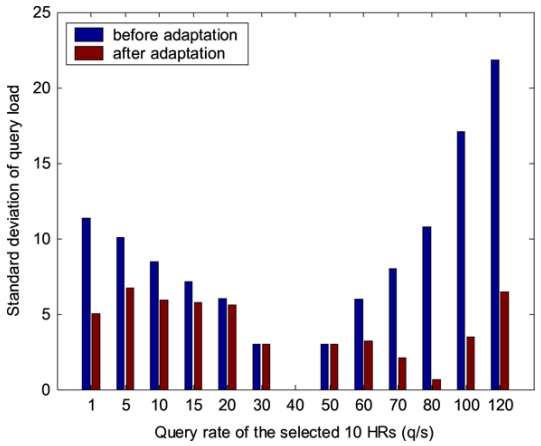

Fig. 9. Standard deviation of query load.

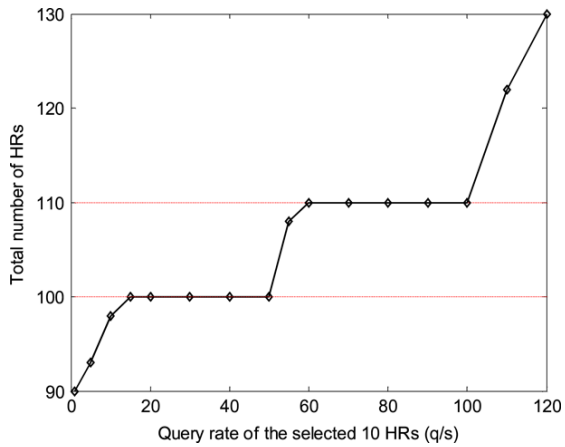

Fig. 10. Number of HRs during adaptation.

Since RCT is designed to be adaptive based on its load status, we design the third experiment to evaluate how RCT adapt in overloaded and light-loaded situations. In practice, the aforementioned thresholds of l_{over}, l_{light} and l_{warning} can be defined as number of quires to process per second. In this experiment, they are set to be 50, 25 and 40 respectively. We let 90 HRs work normally with a load of 40 queries per second. The load of other 10 HRs varies from 1 to 120, which is a process of being light-loaded, normal and overloaded. Fig. 9 shows that the load-aware adaptation mechanism of RCT can balance the query load effectively in both overloaded and light-loaded cases and greatly reduces the risk of system bottleneck. Fig. 10 plots the total number of HRs during adaptation according to the changing load of the selected 10 HRs. When the selected 10 HRs are light-loaded, they will merge with other HRs, and the number of HRs reduces. While in overloaded situation of the 10 selected HRs, new HRs are born to share the load, resulting in an increase of HR number. From Figs. 9 and 10, we can observe that not only load of each HR but also the RCT structure adapts automatically along with load status of HRs.

6. Application to service discovery

In the previous sections, we have taken computational resource discovery as an example to introduce an RCT scheme. Note grid services are another important category of resources in servicing grid systems. This section is devoted to explaining how RCT can be applied to service discovery.

For the metadata description of services, attribute-value pair is a simple and effective method though more complex approaches like ontology technologies exist. We make use of a set of attribute-value pairs to describe grid services. Among the attributes describing a service, there should be an attribute for describing which application domain the service belongs to, e.g. tModels in UDDI. Thus we can categorize services based on their application domain and organizing services of the same application domain into one RCT tree.

However, different from computational resources, most of the attributes of grid services are not numerical, which makes it unsuitable to distribute services of an application domain across tree nodes of an RCT. Therefore in order to apply RCT to service discovery, we need to convert nonnumerical attributes to be numerical ones. This can be done by adoption of Hash functions as DHT-based peer-to-peer technologies do. Compared to DHT schemes, our solution is featured by a categorizing mechanism before routing user request to destination nodes, which results a shorter routing path.

With the above two steps, service discovery can be supported efficiently using RCT scheme. Therefore, we can say RCT can be adopted to build a complete grid information service for service grids.

7. Conclusion

In this paper, we propose an overlay RCT for effective resource discovery in grid. RCT leverages application resource requirement to organize resources based on AVL tree. Although RCT is hierarchical, nodes in higher levels need not maintain more information than those in lower levels, which makes RCT very scalable. RCT is designed to be self-organizing, self-adaptive and fault-tolerant. Commonly used queries such as range queries and multi-attribute queries are well supported by RCT. We conduct extensive evaluations through simulations. Finally, with thorough explanation that RCT can also support service discovery, we conclude that RCT provides a complete solution for grid resource discovery.

Currently, we mainly focus on computational resource discovery and present the basic ideas to apply RCT to service discovery. We will further complete the design of RCT to provide an integrated solution to grid resource discovery. Then we plan to implement the corresponding scheme and deploy the resultant system in real grids like our CROWN grid.

Acknowledgements

This work was supported in part by the 863 programme of China under Grant 2007AA010301, and by the NFSC programme China National Natural Science Funds for

Distinguished Young Scholar under Grant 60525209, partly by National Basic Research Programme of China 973 under Grant 2005CB321803, and partly by the NFSC programme under Grant 60703056. Some preliminary results of this work were presented in the *Proceedings of the* 2nd *IEEE International Conference on e-Science and Grid Computing,* 2006.

References

[1] CROWN Project. http://www.crown.org.cn/en.
[2] DSS application developed in CROWN. http://www.crown.org.cn/en/app/app_info.jsp?appid=110.
[3] A. Andrzejak, Z. Xu, Scalable, efficient range queries for grid information services, in: Proceedings of the 2nd IEEE International Conference on Peer-to-Peer Computing, P2P'02, Linkoping, Sweden, 2002.
[4] A.R. Bharambe, M. Agrawal, S. Seshan, Mercury: Supporting scalable multiattribute range queries, in: Proceedings of ACM SIGCOMM, Portland, OR, USA, 2004.
[5] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, J. Hellerstein, A case study in building layered DHT applications, in: Proceedings of ACM SIGCOMM, Philadelphia, PA, USA, 2005.
[6] A.S. Cheema, M. Muhammad, I. Gupta, Peer-to-Peer discovery of computational resources for grid applications, in: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, Grid 2005, Seattle, WA, USA, 2005.
[7] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman, Grid information services for distributed resource sharing, in: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, HPDC 2001, CA, USA, 2001.
[8] I. Foster, C. Kesselman, J.M. Nick, S. Tuecke, Grid services for distributed system integration, IEEE Computer 35 (6) (2002) 37–46.
[9] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid: Enabling scalable virtual organization, International Journal of Sumpercomputer Applications 15 (3) (2001) 200–222.
[10] J. Gao, P. Steenkiste, An adaptive protocol for efficient support of range queries in DHT-based systems, in: Proceedings of the 12th IEEE International Conference on Network Protocols, ICNP 2004, Berlin, Germany, 2004.
[11] F. Heine, M. Hovestadt, O. Kao, Towards ontology-driven P2P grid resource discovery, in: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, Grid 2004, Pittsburgh, USA, 2004.
[12] J. Huai, H. Sun, C. Hu, Y. Zhu, Y. Liu, J. Li, ROST: Remote and hot service deployment with trustworthiness in CROWN grid, International Journal of Future Generation Computer Systems 23 (6) (2007) 825–835.
[13] A. Iamnitchi, I. Foster, D.C. Nurmi, A peer-to-peer approach to resource location in grid environments, in: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, HPDC 2002, Edinburgh, Scotland, 2002.
[14] M. Li, W.-C. Lee, A. Sivasubramaniam, DPTree: A balanced tree based indexing framework for peer-to-peer systems, in: Proceedings of the 14th IEEE International Conference on Network Protocols, Santa Barbara, CA, 2006.
[15] C. Mastroianni, D. Talia, O. Verta, A super-peer model for resource discovery services in large-scale Grids, International Journal of Future Generation Computer Systems 21 (8) (2005) 1235–1248.
[16] A. Padmanabhan, S. Wang, S. Ghosh, R. Briggs, A Self-Organized Grouping (SOG) method for efficient grid resource discovery, in: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, Grid 2005, Seattle, WA, USA, 2005.
[17] B. Plale, P. Dinda, G.v. Laszewski, Key concepts and services of a grid information service, in: Proceedings of ISCA 15th International Parallel and Distributed Computing Systems, PDCS, 2002.
[18] R. Raman, Matchmaking Frameworks for Distributed Resource Management, Ph.D. Thesis, University of Wisconsin-Madison, 2001.
[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, in: Proceedings of ACM SIGCOMM, 2001.
[20] I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for internet applications, in: Proceedings of ACM SIGCOMM, 2001.
[21] H. Sun, L. Zhong, J. Huai, Y. Liu, OpenSPACE: An open service provisioning and consuming environment for grid computing, in: Proceedings of the 1st IEEE International Conference on E-Science and Grid Computing, e-Science 2005, Melbourne, Australia, 2005, pp. 288–295.
[22] P. Trunfio, D. Talia, H. Papadakis, P. Fragopoulou, M. Mordacchini, M. Pennanen, K. Popov, V. Vlassov, S. Haridi, Peer-to-peer resource discovery in grids: Models and systems, International Journal of Future Generation Computer Systems 23 (7) (2007).
[23] G. Vega-Gorgojo, M.L. Bote-Lorenzo, E. Gomez-Sanchez, A semantic approach to discovering learning services in grid-based collaborative systems, International Journal of Future Generation Computer Systems 22 (6) (2006) 709–719.
[24] X. Zhang, J.L. Freschl, J.M. Schopf, A performance study of monitoring and information services for distributed systems, in: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing, HPDC 2003, Seattle, USA, 2003.
[25] C. Zheng, G. Shen, S. Li, S. Shenker, Distributed Segment Tree: Support of Range Query and Cover Query over DHT, in: Proceedings of the 5th International Workshop on Peer-To-Peer Systems, IPTPS, Santa Barbara, CA, USA, 2006.

Hailong Sun is a researcher in the School of Computer Science and Engineering, Beihang University, Beijing, China. He received his Ph.D. in Computer Software and Theory from Beihang University, and BS degree in Computer Science from Northern Jiaotong University in 2001. His research interests include grid computing, web services, peer-to-peer computing and distributed systems.

Jinpeng Huai is a Professor and Vice President of Beihang University. He serves on the Steering Committee for Advanced Computing Technology Subject, the National High-Tech Program (863) as Chief Scientist. He is a member of the Consulting Committee of the Central Government's Information Office, and Chairman of the Expert Committee in both the National e-Government Engineering Taskforce and the National e-Government Standard office. Dr Huai and his colleagues are leading the key projects in e-Science of the National Science Foundation of China (NSFC) and Sino-UK. He has authored over 100 papers. His research interests include middleware, peer-to-peer (P2P), grid computing, trustworthiness and security.

Yunhao Liu received his BS degree in Automation Department from Tsinghua University, China, in 1995, and an M.A. degree in Beijing Foreign Studies University, China, in 1997, and an M.S. and a Ph.D. degree in Computer Science and Engineering at Michigan State University in 2003 and 2004, respectively. He is now an assistant professor in the Department of Computer Science and Engineering at Hong Kong University of Science and Technology. His research interests include peer-to-peer computing, pervasive computing, distributed systems, network security, grid computing, and high-speed networking. He is a senior member of the IEEE Computer Society.

Dr. Rajkumar Buyya is an Associate Professor and Reader of Computer Science and Software Engineering; and Director of the Grid Computing and Distributed Systems (GRIDS) Laboratory at the University of Melbourne, Australia. He received B.E. and M.E. in Computer Science and Engineering from Mysore and Bangalore Universities in 1992 and 1995 respectively; and Doctor of Philosophy (PhD) in Computer Science and Software Engineering from Monash University, Melbourne, Australia in 2002. He was awarded Dharma Ratnakara Memorial Trust Gold Medal in 1992 for his academic excellence at the University of Mysore, India. He received Richard Merwin Award from the IEEE Computer Society (USA) for excellence in academic achievement and professional efforts in 1999. He received Leadership and Service Excellence Awards from the IEEE/ACM International Conference on High Performance Computing in 2000 and 2003. He received "Research Excellence Award" from the University of Melbourne for productive and quality research in computer science and software engineering in 2005. The Journal of Information and Software Technology in Jan 2007 issue, based on an analysis of ISI citations, ranged Dr. Buyya's work (published in Software: Practice and Experience Journal in 2002) as one among the "Top 20 cited Software Engineering Articles in 1986–2005". He received the Chris Wallace Award for Outstanding Research Contribution 2008 by the Computing Research and Education Association of Australasia, CORE, which is an association of university departments of computer science in Australia and New Zealand.