Combining Efficiency, Fidelity, and Flexibility in Resource Information Services

Haiying Shen, *Senior Member*, *IEEE*, Yuhua Lin, and Ting Li

Abstract—A large-scale resource sharing system (e.g., collaborative cloud computing and grid computing) creates a virtual supercomputer by providing an infrastructure for sharing tremendous amounts of resources (e.g., computing, storage, and data) distributed over the Internet. A resource information service, which collects resource data and provides resource search functionality for locating desired resources, is a crucial component of the resource sharing system. In addition to resource discovery speed and cost (i.e., efficiency), the ability to accurately locate all satisfying resources (i.e., fidelity) is also an important metric for evaluating service quality. Previously, a number of resource information service systems have been proposed based on Distributed Hash Tables (DHTs) that offer scalable key-based lookup functions. However, these systems either achieve high fidelity at low efficiency, or high efficiency at low fidelity. Moreover, some systems have limited flexibility by only providing exact-matching services or by describing a resource using a pre-defined list of attributes. This paper presents a resource information service that offers high efficiency and fidelity without restricting resource expressiveness, while also providing a similar-matching service. Extensive simulation and PlanetLab experimental results show that the proposed service outperforms other services in terms of efficiency, fidelity, and flexibility; it dramatically reduces overhead and yields significant enhancements in efficiency and fidelity.

Index Terms—Resource sharing systems, collaborative cloud computing (CCC), grid computing, resource information service, resource discovery, distributed hash table (DHT)

1 Introduction

A large-scale resource sharing system (e.g., collaborative cloud computing and grid computing) creates a virtual supercomputer by providing an infrastructure for sharing tremendous amounts of resources over the Internet. Grid computing has dramatically evolved from its roots in science and academia, and is currently at the onset of mainstream commercial adoption. With the tremendous development of cloud computing [1], collaborative cloud computing (CCC) [2], [3] has been proposed to connect a large number of clouds as an alliance that come together to share resources in order to better respond to large-scale application requirements. CCC can handle the situation when a single cloud is not sufficient to provide sustainable high-quality service to some applications with demand for scalable resources or when researchers want to build a virtual lab environment across geographical distribution of physical hosts [2]. For example, cloud customer Dropbox had around 100 million users in 2012, and around 50 million users in 2011, which is three times the number of 2010. As a cloud may be overloaded during peak periods and stay idle in time periods with few service requests, it is promising to integrate many dispersed clouds from different corporations and organizations to fully utilize cloud resources.

- *H. Shen and Y. Lin are with the Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634. E-mail: {shenh, yuhual}@clemson.edu.*
- *T. Li is with Wal-mart Stores Inc., Bentonville, AR 72716. E-mail: dragonflyting@hotmail.com.*

Manuscript received 23 Apr. 2013; revised 27 Aug. 2013; accepted 04 Nov. 2013. Date of publication 19 Nov. 2013; date of current version 16 Jan. 2015. Recommended for acceptance by T. Gonzalez.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.222

The large-scale resource sharing system makes possible the sharing of a variety of resources including CPU time, storage, memory, network bandwidth, software, data (books, music, videos) and devices distributed over a wide area. A computing resource (e.g., virtual machine) is described by a set of attributes such as CPU speed, memory, OS version and device name. A data resource also can be described by a few keyword attributes. For example, if a node (i.e., physical machine) needs a resource with attributes "$\mathrm{OS\ name} = \mathrm{Linux}$", "$\mathrm{CPU\ speed} = 1000\ \mathrm{MHz}$", and "$\mathrm{Free\ memory} = 1024\ \mathrm{MB}$" for a computing task, how can it quickly discover the required resources with low overhead?

Since resource discovery is an indispensable function in large-scale resource sharing systems, resource information services are a crucial component, as they collect resource data and provide resource search functionality in order to bridge resource providers and requesters. However, an effective resource information service must meet three challenges. The first challenge is achieving high efficiency in an environment characterized by large-scale, geographically scattered resources and dynamics. In such an environment, millions of heterogeneous resources are scattered across geographically distributed nodes, resource utilization and availability are continuously changing, and nodes enter or leave the system unpredictably [2], [4]. The second challenge is guaranteeing the high fidelity of resource location. Fidelity means the ability to locate all resources in the system that satisfy a resource request. It is defined as the true positive rate of the located resources, i.e., the total number of satisfying resources located divided by the total number of satisfying resources in the system. A method with higher fidelity misses fewer satisfying resources in the system. The third challenge is achieving flexibility. Flexibility means the ability to allow nodes to

specify their desired resources with unlimited expressiveness, and to conduct similar-resource searching rather than exact-matching searching. A resource information service lacks flexibility if it predefines the attributes for use in resource discovery. Similar resources are resources with similar resource descriptions. Resource descriptions with more common attributes have higher similarity.

Current resource discovery approaches in clouds [5]–[8] and traditional approaches for grids operate in a centralized manner [9]–[12]. Resource providers report their available resource information to a resource information server (one or several brokers), and then resource requesters contact the server for their desired resources. These centralized approaches are insufficiently scalable due to bottlenecks and single point of failure. The information server can be overloaded in a large-scale system, leading to service inefficiency; in turn, the entire resource discovery mechanism will fail.

To handle the high efficiency and scalability challenge in the large-scale resource sharing systems, distributed Hash Tables (DHTs) [13]–[18] provide an attractive solution for resource information services with their inherent properties of high scalability, self-organization and fault-tolerance in large-scale distributed systems. However, most previous DHT-based resource information service systems either achieve high fidelity at low efficiency or achieve high efficiency at low fidelity. Moreover, the systems have limited flexibility by only providing exact-matching services or describing a resource using a pre-defined attribute list.

One group of DHT-based systems [19]–[23] separate the attributes of a resource and map the resource into a DHT overlay based on each attribute. We call this group of resource information systems "direct mapping". Since all resource information containing a specific attribute is stored in one node, the approaches result in load imbalance among nodes, and lead to high costs for resource searching among a huge volume of information. When a requester searches for a resource, it searches for each attribute of the resource and then merges the results. For an m-attribute resource, these approaches need m pooling messages, memory for storing m pieces of information, and m messages for querying the resource. In addition, the requester receives all resource data pertinent to an attribute in its query, and needs to derive the information of required resources with a merging operation, despite the fact that the requester might be interested in only a subset of the data. Though these approaches have high fidelity, they incur a high overhead for resource pooling, searching, and merging. Also, direct mapping systems only provide an exact-matching service, though it is often more appropriate for a user to formulate search requests in less precise terms, rather than defining a sharp limit. A similar-matching service not only provides this flexibility but also helps to reduce overhead, since a requester only receives the information of similar resources.

Another group of resource information systems [24] combines all attributes of a resource into a single key, and then maps the resource to the key's owner in a DHT overlay. We call this system "one-point mapping". For an m-attribute resource, this system needs one pooling message, memory for storing one piece of information, and one message for querying the resource. Unlike direct mapping, this system generates much lower overhead. However, converting a number of attributes into a single key may not accurately preserve the similarity between the sets of attributes, especially when there are a large number of attributes in a resource. Thus, two similar resources may create totally different keys. As a result, many similar resources cannot be located in response to a request. One-point mapping systems offer high efficiency at the cost of low fidelity.

Recently, we proposed PIRD [25] resource discovery mechanism, which weaves all attributes into a set of indices using locality sensitive hashing (LSH) [26], [27] and then maps the indices to a DHT overlay. It reduces overhead and yields significant improvements in the efficiency of resource discovery. Both PIRD and the one-point mapping system have limited flexibility with restricted expressiveness. They build a multi-dimensional space with each attribute as a coordinate. Users' requested resource attributes are confined to predefined resource attributes, impeding the flexibility of new resource attribute creation.

This paper presents an LSH-based resource information Service (LIS) that combines efficiency, fidelity, and flexibility. LIS offers high efficiency and fidelity without restricting resource expressiveness, while providing a similar-matching service. It aims to achieve high efficiency, fidelity and flexibility. We propose three algorithms to transform a resource description to a set of integers. We further build LSH functions by combining the algorithms with min-wise independent permutations [28]. The hash functions generate a set of IDs for a resource that preserves the similarity of resources without requiring a pre-defined attribute list. Based on the generated IDs, LIS stores the resource information into a DHT overlay. Relying on the DHT object location protocol, requested resource information can be efficiently discovered. LIS also offers flexible resource expressiveness and a similar-matching service that enables users to discover similar resources. LIS also incorporates a load balancing algorithm to balance the load for storing resource information and processing resource requests. Note that LIS can be applied to large-scale resource sharing systems that connect large-scale distributed resources for sharing resources including grids and CCC.

The rest of this paper is structured as follows. Section 2 presents a concise review of representative resource discovery methods. Section 3 presents an introduction to DHT overlays, LSH construction, and LIS. Section 4 shows the performance of LIS in comparison to other systems using a variety of metrics and analyzes the factors affecting information service quality. Section 5 concludes this paper with remarks on future work.

2 Related Work

DHT overlays [13]–[15] provide an attractive solution for large-scale resource information services due to their scalability, reliability, and dynamic-resilience. Many DHT-based resource information services have been previously proposed for grids. In DHT-based direct mapping systems, some systems adopt one DHT for each attribute, and process resource queries at the same time in corresponding DHTs [21], [22]. Thus, if a system has m resource attributes, the information service needs m DHT overlays. Depending on multiple DHTs for resource discovery leads to high maintenance overhead for DHT structures. Another group of resource information services [19]–[23], [29], [30] organizes all resource information into one DHT overlay. In this group, attributes of a resource

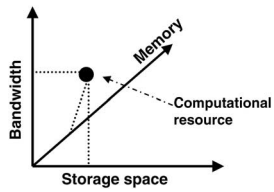


Fig. 1. A 3-D keyword space.

are separated, and the resource information of each attribute is pooled in a DHT node. This strategy results in load imbalance among nodes, and leads to high costs for searching among a huge volume of information in a single node. In most direct mapping systems, when a requester searches for a resource, it searches for each attribute of the resource and then merges the results. As indicated previously, these methods generate high costs for information storage and location and only provide an exact-matching service. Also, they provide high fidelity, but at the cost of low efficiency.

Schmidt and Parashar [24] proposed a dimension reducing indexing scheme for resource discovery. They built a multi-dimensional space with each coordinate representing a resource attribute. Fig. 1 shows an example of a 3-dimensional keyword space. The resources are viewed as base-r numbers, where r is the total number of attributes in the grid system. This method projects a multi-attribute resource to a point in the multi-dimensional space. The dimension reducing indexing scheme then transforms the multidimensional resource attribute to one dimension using a space-filling curve (SFC) [31], while still preserving the similarity between the resources. The system then maps the resource point to a DHT node. This guarantees that all existing resources that match a query are found with bounded costs in terms of the number of messages and nodes involved. However, the scheme is not effective in discovering satisfying resources when there are a large number of attributes because of the degrading performance of SFC dimension reduction in a high dimensional space. This method offers high efficiency but at the cost of low fidelity.

Our previous work [25] proposed PIRD, a DHT-based resource discovery mechanism in Internet-based distributed systems. PIRD builds a multi-dimensional space as in [24]. A resource has a vector, the size of which is the number of dimensions. PIRD relies on an existing LSH technique in Euclidean spaces [32] to create a number of IDs for a resource, and then maps the resource to DHT nodes. In a system with a tremendous number of resource attributes, PIRD leads to dramatically high memory consumption and low efficiency of resource ID creation due to long resource vectors. Recognizing this drawback, we further developed optimized PIRD (OPIRD), using the LZW dynamic compression algorithm [33] to reduce the size of resource vectors. However, the compression comes with extra cost for conducting the algorithm.

Since one-point mapping [31] and PIRD build a pre-defined attribute list, they are not sufficiently flexible in dealing with new attributes. To overcome this problem, our proposed LIS builds new LSH functions to transform resources to resource IDs, which does not require a pre-defined attribute list. Thus, LIS significantly reduces memory consumption and improves the efficiency of resource ID creation. More importantly, LIS ensures fidelity for discovering resources in an environment with a significant number of resource attributes.

There are a few works for resource discovery in clouds. Nordin et al. [5] used a cloud resource broker for goal-based requests in medical applications. The resources are defined as data storage, computer operating system, CPU speed, etc. The broker discovers satisfying machines and allocates them to users based on their different quality of service (QoS) requirements in order to maximize/optimize allocation of resources to the user's desired goal(s). Goscinski et al. [6] used a number of brokers to map resource/service providers and clients. The resource providers report their dynamic attributes (i.e., current state and characteristic of cloud services and resources) to the brokers. Clients tell the brokers their preferences for resources/services. The brokers then publish the information of satisfying resources/services from the providers to the clients. Sun et al. [7] proposed a model for discovering cloud resources in a multi-provider environment, which hides the complex implementation details from the resource providers. In this model, cloud resources are described with different attributes such as hardware specification and storage capacity, client requirements are translated into a set of constrains, and a client request is mapped to a set of resources. Yan et al. [8] produced a distributed content-based publish/subscribe system for resource discovery in clouds. In this system, resource providers act as publishers, resource discovery clients act as subscribers, and brokers push providers' messages containing satisfying resources to the clients. These cloud resource discovery methods that rely on broker(s) are not scalable for large-scale resource discovery due to their centralized manner, while the distributed manner and high-scalability of our DHT-based LIS enable it to meet the requirements of the large-scale resource sharing environment.

3 Efficient, Flexible, and High-Fidelity Resource Information Service

The goal of LIS is to provide a resource information service with low overhead and high efficiency, fidelity, and flexibility. A critical function of the service is to translate resource descriptions into IDs (which are numerical values) in a locality preserving manner. That is, the same resource descriptions are translated to the same IDs and similar resource descriptions are translated to close IDs, facilitating similar resource searching. If $|ID_j - ID_i| < |ID_k - ID_i|$, we say that ID_j is closer to ID_i than ID_k. Unlike the previous systems, LIS does not need to form a unified multi-dimensional space consisting of all resource attributes for resource translation. Thus, it avoids the complexity of maintaining such a multi-dimensional space, and reduces the memory for storing long resource vectors. More importantly, it avoids the curse of dimensionality [26] and can preserve higher locality in resource translation. Specifically, we propose three algorithms that can transform a resource attribute to an integer in a similarity preserving (i.e. locality preserving) manner. We then develop LSH functions by combining the algorithms with min-wise independent permutations. The hash functions translate a resource description to a set of IDs, which help to store and query the information of the resource in a DHT overlay. The DHT overlay is formed by all nodes in the large-scale resource sharing system.

3.1 Resource Attribute Transformation

Note that a resource description consists of a set of attributes (including values). To transform each attribute of a resource

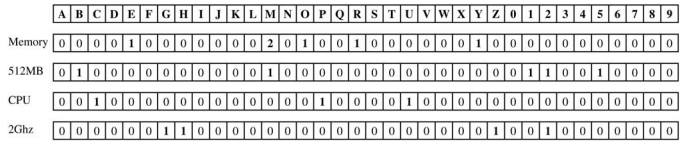


Fig. 2. The *Alphanumeric* attribute transformation.

into an integer, we propose three attribute transformation algorithms with similarity preserving features for each attribute. Consistent hash functions such as SHA-1 [34] are widely used in DHT networks to generate node or file IDs due to their collision-resistant nature. Using this hash function, it is computationally infeasible to find two different messages that produce the same message digest. The first attribute transformation algorithm, denoted by *SHA*, applies a consistent hash function on each attribute of a resource to change the resource description to a resource vector. For example:

resource description A: Memory 512 MB CPU 2 GHz
resource vector A': 1945 6281 214 2015.

Because of the collision-resistant nature of the consistent hash function, the same attributes will be transformed into the same integers. Thus, *SHA* preserves the similarity between resource descriptions to a certain extent. However, the *SHA* algorithm can only offer exact attribute matching service, but cannot provide similar attribute matching service. For example, it generates different values for 'MEM" and "Memory".

The second attribute transformation algorithm, denoted by *Alphanumeric*, relies on an alphanumeric list to change a resource attribute into an integer. As shown in Fig. 2, each resource attribute is represented by a 36-bit binary number. To determine the integer of a resource attribute, the attribute is mapped to the alphanumeric list. If the attribute does not contain the letter or digit in a bit position of the list, it has 0 in this position. If the attribute contains the letter or digit in a bit position of the list, it has the number of occurrences in this position. For example, the resource vector of "Memory 512 MB CPU 2 GHz" is:

000010000000201001000001000000000000
010000000000100000000000000110010000
001000000000000100001000000000000000
000000110000000000000000010010000000.

This algorithm provides a certain degree of attribute similarity preserving. For example, "MEM" and "Memory" will have similar values. However, since the alphanumeric transformation works by counting the frequency of alphabetic and numerical characters in a resource attribute, the attributes with similar character frequency (e.g., 112 and 211) would be regarded as similar attributes. Note this alphanumeric transformation only provides an approximation method in preserving the closeness of resource descriptions. LIS has a final filtering operation to filter out false positives in discovered resources. Also, if the difference of integers is used to evaluate the degree of similarity between resources, the characters in higher significant bit positions have higher weights. We propose the third algorithm that gives the characters in the alphanumeric list the same importance when checking the similarity between resources. Note that value attributes actually need to distinguish the weights of different digits. The alphanumeric list already regards the digits as numerical characters and fills them into the numerical character list. Then, we also rely on the final filtering operation to filter out false positives caused by the different weights of numerical characters in numerical attributes. The third algorithm combines the *Alphanumeric* algorithm with a Hilbert space-filling curve (SFC) technique [31], denoted by *Hilbert*. The alphanumeric list can be regarded as a 36-dimension Cartesian space with each letter or digit representing a dimension. SFC maps points in the 36-dimensional Cartesian space into a domain of real numbers such that the closeness relationship among the points is preserved. This mapping can be regarded as filling a curve within the m-dimensional space until it completely fills the space. The m-dimensional space is partitioned into 2^{mx} grids of equal size (x controls the number of grids used to partition the m-dimensional space), and each point is numbered according to the grid into which it falls. Briefly, *Hilbert* applies the Hilbert SFC hash function to the integer generated by the *Alphanumeric* algorithm. For example:

resource description A: Memory 512 MB CPU 2 GHz
resource vector A': 19 61 24 15.

Since the same resource attributes are transformed into the same integers, and more similarly described attributes (e.g., Mem and memory) are transformed to closer integers, *Hilbert* provides similarity-preserving resource attribute transformation. In addition, *Hilbert* has better similarity preserving capability than *Alphanumeric* because it assigns the same weight to all bits in the intermediate 36-bit number when generating the final integers. The attribute transformation algorithms treat all attributes equally, and we leave the consideration of the different importance of attributes to our future work.

3.2 Locality-Sensitive Resource Translation

Using the introduced attribute transformation algorithms and min-wise independent permutations [28], we develop new LSH functions that do not need a pre-defined attribute list. We use SHA-LIS, Alpha-LIS and Hilbert-LIS to represent the new LSH functions combining *SHA*, *Alpha* and *Hilbert* with min-wise independent permutations respectively.

In the following, we first introduce the definitions of *locality sensitive hash functions* and *min-wise independent permutations*. We then explain how to use min-wise independent permutations to construct a new LSH function. Finally, we explain how to use the attribute transformation algorithms introduced in Section 3.1 to transform resource attributes to values, which then enable the operation of the min-wise independent permutations. As a result, the new LSH function can be directly applied to a resource description to translate it to hash values (i.e., IDs).

Definition 1. *A family of hash functions $\mathcal{F}$ is said to be a locality sensitive hash function family corresponding to similarity function $sim(A, B)$ if for any two sets A and B from the domain of hash functions, we have:*

$$\mathbf{Pr}_{h\in\mathcal{F}}(h(A) = h(B)) = sim(A, B), \tag{1}$$

where $\mathbf{Pr}$ is probability and $sim(A, B) \in [0, 1]$ is a similarity function [35], [36].

Definition 2. *Let $[n]$ be a list of n numbers and S_n be the set of all permutations of $[n]$. $\mathcal{F} \subseteq S_n$ is min-wise independent if for any set $X \subseteq [n]$ and $x \in X$, when permutation π is chosen at random in $\mathcal{F}$,*

$$\mathbf{Pr}(min\{\pi(X)\} = \pi(x)) = \frac{1}{|X|}, \tag{2}$$

where $min\{\cdot\}$ is the minimum function that selects the minimum value in $\{\cdot\}$. In other words, all the elements of any fixed set X have an equal chance of becoming the minimum element of the image of X under π [28].

Min-wise independent permutations provide an elegant construction of an LSH function with the Jaccard set similarity measure:

$$sim(A, B) = \frac{|A \cap B|}{|A \cup B|}. \quad (3)$$

For example, the similarity between A and B, where
A = Memory|512 MB|CPU|2 GHz, and
B = Memory|512 MB|CPU|3.5 GHz, is $\frac{3}{5} = 0.6$.
By combining formulas (2) and (3), we get:

$$\mathbf{Pr}_{h\in\mathcal{F}}(min\{\pi(A)\} = min\{\pi(B)\}) = \frac{|A \cap B|}{|A \cup B|} = sim(A, B).$$

Based on Definition 1, $min\{\pi(A)\}$ is an LSH function.

As the work in [28], LIS defines the min-wise independent permutations as:

$$\pi(x) = (ax + b) \bmod \hat{p},$$

where a and b are random integers, $0 < a \leq \hat{p}$ and $0 \leq b \leq \hat{p}$, and $\hat{p}$ is a large prime number that is greater than the hash value space of the attribute transformation algorithms. Because the permutations can only be performed on values rather than attributes but a resource description consists of attributes, we combine an attribute transformation algorithm and the min-wise independent permutations to build a locality hash function. Given a resource description A, using one of the previously introduced attribute transformation algorithms, LIS converts A into $A' = \{a_1, a_2, \ldots, a_l\}$, where $a_i (1 \leq i \leq l)$ is an integer and l is the number of attributes in the resource description. An LSH function h_π is constructed as:

$$h_\pi(A') = min\{\pi(A')\} = min\{\pi(a_1), \pi(a_2), \ldots, \pi(a_l)\}.$$

The hash result is the minimum of the permutation results of the elements in A'. Given two resource descriptions A and B, we first convert them to A' and B'. Then, $x = h_\pi(A') = h_\pi(B')$ iff $\pi^{-1}(x) \in A' \cap B'$ where $\pi^{-1}(x)$ represents the inverse of x. $x = h_\pi(A') = h_\pi(B')$ also means that $x = h_\pi(A' \cup B')$. Because π is a random permutation, each element in $A' \cup B'$ is equally likely to be $h_\pi(A' \cup B')$. Then, we can get that $h_\pi(A') = h_\pi(B')$ occurs with probability $sim(A', B') = \frac{|A' \cap B'|}{|A' \cup B'|}$. Recall that the attribute transformation algorithms preserve the similarity between A and B after they are converted into A' and B' respectively, then the probability $p = sim(A, B) = sim(A', B') = \frac{|A' \cap B'|}{|A' \cup B'|}$. Therefore, h_π is an LSH function for resource vectors since

$$\mathbf{Pr}(h_\pi(A') = h_\pi(B')) = sim(A', B').$$

Let's use an example to see how h_π transforms a resource to a hash value. Given a resource description $A =$ "Memory 512 MB CPU 2 GHz", an attribute transformation algorithm translates the resource description to resource vector "19, 61, 24, 15". Then, hash function h_π is applied to the vector. That is,

$$\begin{aligned} h_\pi(A') = h_\pi(19, 61, 24, 15) &= min\{\pi(19, 61, 24, 15)\} \\ &= min\{\pi(19), \pi(61), \pi(24), \pi(15)\}, \end{aligned}$$

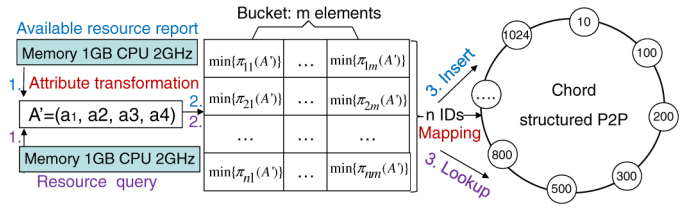


Fig. 3. Process of the LIS operation.

where $\pi(x) = (ax + b) \bmod \hat{p}$. If $a = b = 1$,

$$h_\pi(A') = min\{20, 62, 25, 16\} = 16.$$

Algorithm 1 Pseudo-code for locality-sensitive resource translation

n.**generateID** () {

//build min-wise independent permutations

Generate n groups of m random integers a and b

//transform attributes to integers

for each attribute in a resource description **do**

Use an attribute transformation algorithm to

transform the attribute to integer $k[i]$

endfor

//generate $n \times m$ hash values for a resource using $h_\pi \min[p][q] = 0$

for each $a[p][q]$ and $b[p][q]$ **do**

for each $k[i]$ **do**

$g = (a[p][q] * k[i] + b[p][q]) \bmod \hat{p}$

$min[p][q] = min(min[p][q], g)$

endfor

endfor

//generate n IDs for a resource using XOR

for $i = 0$ to n **do**

for each $min[i][j]$ **do**

$\mathrm{ID[i]}^{\wedge} = min[i][j] mod\, ID_{p2p}$ *//$^{\wedge}$ is a XOR operation*

endfor

endfor

}

Fig. 3 illustrates the process of the LIS operation. To build a family of hash functions $\mathcal{F}$, LIS makes n groups of hash functions with each group having m numbers of h_π. Applying the $m \times n$ hash functions to a resource vector A', we get n buckets with each bucket having m hash values. A bucket represents a group of m hash values generated by $h_{\pi_i} (1 \leq qi \leq m)$. LIS then executes an XOR operation on the values in each bucket to obtain a final hashed value. Consequently, each record has n numerical hash values, denoted by *IDs*.

$$ID_i = (min\{\pi_1(A')\}^{\wedge} \ldots min\{\pi_m(A')\}) \bmod ID_{p2p},$$

where $1 \leq i \leq n$ and ID_{p2p} is the length of the P2P key space. Similar groups of $h_{\pi_i}(A')\,(1 \leq i \leq m)$ are transformed to close IDs by XORing their m hash values; hence, the XOR operation preserves the similarity in this hashing step. Algorithm 1 shows the pseudo-code for the locality-sensitive resource translation in LIS. Consequently, similar resources have close IDs. The IDs of a resource are its final keys for storing and searching the resource information in a DHT overlay.

In the one-point mapping and PIRD resource information systems, the size of each resource vector equals the number of all resource attributes in the system before hashing. It leads to high memory consumption and inefficiency for hashing. In contrast, LIS does not require that all resource vectors have the same dimension, so the size of a record's vector equals the number of attributes in the resource description. Thus, LIS reduces memory consumption and enhances efficiency. On the other hand, the direct mapping resource information systems apply a consistent hash function to each resource attribute in order to generate an integer in the resultant resource vector, and then map the resource to a DHT overlay based on each integer. This method generates many messages for storing and searching resources that consist of a large number of attributes. Also, merging many located resources for satisfying resources generates high overhead. LIS maps a resource to n nodes regardless of the resource vector size while enabling users to search for resources similar to the queried resource.

3.3 DHT-Based Resource Information Service

LIS is built on top of a DHT overlay formed by all nodes in the system to achieve multi-attribute resource searching. In the above sections, we introduced how to translate resources to IDs. Using these resource IDs, resource providers store the information of their available resources to the DHT, and resource requests will be forwarded to the DHT nodes storing the information of the required resources. Thus, the DHT functions as a matchmaker between resource providers and requesters.

A DHT overlay is a decentralized system in the application level that partitions ownership of a set of objects among participating nodes and can efficiently route messages to the unique owner of any given object. DHTs have maintenance mechanisms to handle node joins and voluntary departures. However, if an owner fails without warning, the objects stored in the owner are lost. Object replication is a strategy to handle this problem, in which an object is stored in multiple nodes to avoid losing objects [37]–[39]. We use Chord [13] as a representative of DHT overlays to explain the LIS resource information service. Chord achieves a time complexity of $O(\log N)$ per lookup request by using $O(\log N)$ neighbors per node, where N is the number of nodes in the overlay. Each object or node is assigned an ID (i.e., key) that is the hashed value of the object name or node IP address using a consistent hash function [34]. An object's successor node is the node whose ID equals or immediately succeeds the object's ID. An object is assigned to its successor node in the ID space. For example, in Fig. 3, if two objects have IDs 90 and 100, they will be stored in node $n100$. The overlay network provides two main functions: `Insert(key, object)` and `Lookup(key)`, to store an object to a node responsible for the key and to retrieve the object, respectively. The message for the two functions is forwarded from node to node based on the DHT routing protocol through the overlay network until it reaches the object's owner.

We represent the resource predicate of a resource in the form of $< v, ID, ip\_addr >$, where v denotes the resource description vector and ip_addr is the IP address of the resource owner. By using the resource IDs as DHT keys, the resource predicate of a resource is indexed into the DHT overlay. For instance, using our developed LSH function, a node produces n IDs for its resource "Memory 512 MB CPU 2 GHz Bandwidth 10 Mbps", $ID_2, \ldots, ID_n$. It then uses

$$\mathrm{Insert(ID_i,\ < v, ID_i, ip\_addr >)}\,(1 \leq i \leq n)$$

to insert the predicate of the resource to the DHT overlay. Consequently, the resource predicate is stored in n nodes in the DHT overlay.

Algorithm 2 Pseudo-code for storing resource predicates

```
n.storeResc () {
    generateID();
    for each ID[i] do
        Executes Insert(ID[i], < v, ID[i], ip_addr >)
    endfor
}
```

In LIS, each node periodically inserts the predicates of its available resources into the DHT system. Due to the similarity-preservation of our developed LSH functions, the predicates of similar resources will be stored in the same or close nodes. This facilitates similar resource searching in the resource information service. Algorithm 2 shows the pseudo-code for storing resource predicates in LIS. Note that for two resource vectors A' and B',

$$\mathbf{Pr}_{h \in \mathcal{H}}[h(A') = h(B')] = sim(A', B') = p.$$

This resource predicate storing method can store the predicates of similar resources in the same nodes with probability $\geq 1 - (1 - p^m)^n$ [32]. The probability that two resource vectors do not have the same ID after the operation on one bucket is $1 - p^m$. The probability that they do not have the same ID after the operations on all n buckets is $(1 - p^m)^n$. Thus, the probability that they have at least one same ID is $1 - (1 - p^m)^n$.

A resource query is also represented by a resource description. For example, a typical query "Memory 512 MB CPU 2 GHz Bandwidth 10 Mbps" specifies a computer resource with 512 MB memory, 2 GHz CPU and 10 Mbps bandwidth. The expected results of a query are the resource predicates of a complete set of resources that match the user's query. The DHT lookup protocol guarantees that an object can be located in $O(\log N)$ hops.

Algorithm 3 shows the pseudocode of resource querying in LIS. For example, let A' be a query's resource vector. Resolving the query means locating the resources whose vectors are similar to A'. LIS first produces n IDs from A' using the our LSH function. Note that the query and the vectors of similar resources are hashed to the same IDs with high probability (i.e., $1 - (1 - p^m)^n$). Thus, by using these IDs as the DHT keys in the function $Lookup\,(ID_i)(1 \leq i \leq n)$, the resource requester will receive the predicates of its desired resources from the

destination nodes. This means that a query can be answered by only consulting a small number (i.e., $\leq n$) of nodes.

Algorithm 3 Pseudo-code for resource querying

```
n.discoverResc () {
    generateID();
    for each ID[i] do
        Executes lookup(ID[i])
    endfor
    Receive replies from all destination nodes
    Merge the predicates in all responses that satisfy the query
    Request resources from resource owners
}
```

Assume $n = 3$ and resource "Memory 512 MB CPU 2 GHz" has IDs 10, 200, and 500 after applying our developed LSH function. In Fig. 3, the resource predicate of the resource will be stored in nodes $n10$, $n200$, and $n500$ by `Insert()`. When a node needs resource "Memory 512 MB CPU 2 GHz", it first generates the IDs of the resource using our developed LSH function, which are 10, 200, and 500. It then sends requests `Lookup()` using the IDs as targets. The three requests will be forwarded to nodes $n10$, $n200$, and $n500$ which will then respond to the resource requester of its queried resources.

Since similar resources have close IDs, the nodes close to the destination nodes can be searched in order to avoid missing similar resources. Thus, a range value, R, is determined and the nodes with IDs in $[ID - R, ID + R]$ are queried during resource searching. Specifically, when a node receives a query, it forwards the query to its successor and predecessor, which will further forward the query to their successor and predecessor respectively, until the node ID is no longer in the range $[ID - R, ID + R]$. In the example, if $R = 2$, then the nodes whose IDs are in [8], [12], [198,202], and [498,502] will be searched. After node $n10$ receives the query, it forwards the query to node $n9$ and $n11$. $n9$ further forwards the query to $n8$, and $n11$ further forwards the query to $n12$. Each query receiver sends the information of resources with IDs in $[ID - R, ID + R]$ to the resource requester.

After receiving responses from all destination nodes, the requester merges the replies. In order to remove unsatisfying resources, the requester executes a filtering operation. It calculates the similarity between each received resource and the queried resource. The resources whose similarities are less than the threshold specified by the resource requester are removed. Finally, the requester chooses the resources matching its requirements.

LSH has two important parameters: m and n. m represents a tradeoff between the time spent computing hash values and the time spent filtering false positives in returned information. Given m, an optimal value of n is found that ensures that the number of false positives is no more than a user specified threshold [40]. Larger m values require more memory for hash tables and more time for computing hash values but lead to fewer false positives and hence shorter filtering time and vice versa. It is desirable to set a value for m that reduces memory consumption and hash value computing time and meanwhile reduces the time for filtering false positives. For more details about the tradeoff performance, please refer to [41].

We can further consider the resource cost in the resource information service in the large-scale resource sharing systems, where a node needs to pay for the resources from another node it consumes. We can also take into account the resource quality and location in the resource information service as in [42]. The quality of a resource is measured by the reputation feedback of the consumer of this resource. When a node reports its available resources, it adds the resource cost and location into the resource predicate. When a node requests for a resource, it specifies the requirements on cost, quality and location. Then, when the destination node receives the resource request, it only returns the resource information matching the requester's specified requirements.

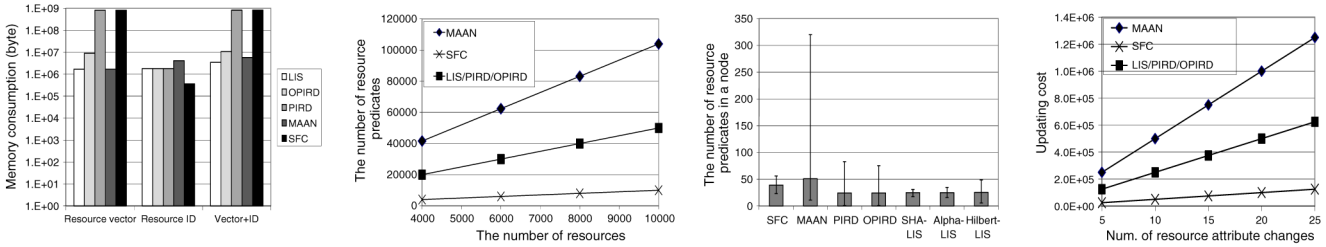
3.4 Load Balancing

The resource predicates are distributed among nodes based on their IDs. Each node is responsible for handling the resource queries for the resource predicates stored in the node. Thus, each node's overhead for handling resource queries is proportional to the number of resource predicates stored in the node. Though the capacity metric of a node in practice should be a function of the node's storage space, bandwidth, CPU, and so on [15], [43], as the works in [44]–[48], we assume that there is only one bottleneck resource for optimization and that each node devotes a certain capacity for handling resource queries, denoted by C. We use L to denote the overhead caused by received resource queries. To avoid overloading any node, we aim to achieve $L < C$ for every node via a load balancing algorithm.

Normally, in a load balancing algorithm, each node periodically measures its load. If it is lightly loaded (i.e., $C > L$), it calculates its extra capacity $\Delta C = C - L$, and if it is overloaded (i.e., $L > C$), it calculates its extra load $\Delta L = L - C$. Each node periodically reports its load status (i.e., ΔC or ΔL) to one repository node or a number of repository nodes. The repository node(s) match the ΔC and ΔL and notify corresponding heavily loaded nodes to move the load (i.e., stored resource predicates) to matched lightly loaded nodes.

A load balancing algorithm with only one repository node is a centralized algorithm. As the centralized repository node can easily become a bottleneck, centralized algorithms are not suitable for large-scale resource sharing systems. We thus employ a decentralized algorithm that uses all nodes in the system as the repository nodes for load balancing. Specifically, nodes report their load status in the bottom-up fashion of a tree. Each information receiver (i.e., parent) conducts matching between ΔC and ΔL, and reports unresolved ΔL and remaining ΔC to its parent. As the load status information flows upwards, the extra load from heavily loaded nodes is moved to lightly loaded nodes. Finally, the tree root conducts the matching and the entire system reaches load balance.

Suppose the ID space of the Chord DHT is $[1, N]$. The tree root is the node whose ID is closest to $\frac{N}{2}$. Then, the ID space is partitioned into two parts $[1, \frac{N}{2}]$ and $[\frac{N}{2} + 1, N]$. Each part is partitioned into k parts, and the nodes whose IDs are closest to the middle point of each partition become the children of the root. This process repeats until each partition has no more than k



(a) Memory consumption (b) Num. of resource predicates (c) Predicate distribution balance (d) Resource predicate updating cost in attribute dynamics

Fig. 4. Overhead of different resource information services.

IDs. In this way, each node can calculate the ID closest to its parent's ID. Through the DHT *lookup(ID, loadStatus)* function, a node can send its load status to its parent. After a parent receives the load status information, it orders the ΔC and ΔL information in a descending order, fetches each ΔL, and finds ΔC that is no less than ΔL. It then notifies the node of ΔL to move its extra load to the node of ΔC. The child-parent forwarding operations complete by $\log N$ steps in the tree. The load balancing operation stops when the root node completes the matching process and the algorithm is executed periodically.

After a heavily loaded node i moves its extra load (i.e., stored resource predicates) to a lightly loaded node j, node i creates an index indicating the resource predicate IDs and the IP address of node j. When node i receives resource queries for the moved resource predicates, it forwards the queries directly to node j using j's IP address. Node j then handles the queries and responds to the resource requesters. When node j becomes heavily loaded later on, it may further move the resource predicates from node i to a lightly loaded node, say node k, in the load balancing operation. In this case, node j notifies node i to update its index to point to node k. Thus, node i that originally stores the moved resource predicates always maintains an index pointing to the final location of these resource predicates, which generates only one extra step in the resource querying process.

4 Performance Evaluation

We designed and implemented a simulator for the evaluation of the LIS resource information service. We compared LIS with a direct mapping system, a one-point mapping system, and PIRD/OPIRD [25] in terms of overhead, efficiency, and fidelity. We used MAAN [19] as the direct mapping system and SFC [24] as the one-point mapping system.

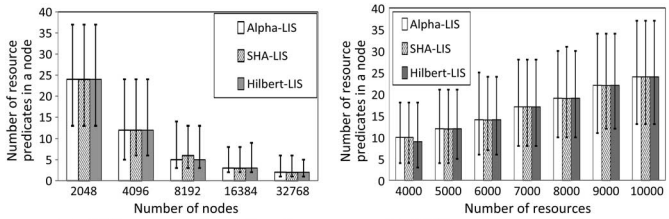
The number of nodes in Chord was set to 2,048. The total number of attributes in the system was set to 20,591, and the total number of resources was set to 10,000. The number of resource queries was set to 100 unless otherwise specified. We set $m = 20$ and $n = 5$ in LIS unless otherwise specified. These values achieve relatively short time for both hash value computing and filtering false positives. The number of attributes in a resource description and a resource query was set to 9. We include some of the experimental results in [25] for reference.

4.1 Overhead

Resource predicates stored in a node include resource vectors and IDs. Fig. 4(a) shows the total memory consumption in bytes for resource vectors and IDs. We see that SFC and PIRD consume prohibitively more memory for resource vectors. They regard each resource attribute as a dimension in a multi-dimensional space, which makes the size of a resource vector equal the number of total attributes in the system. OPIRD reduces the resource vector size by using a compression algorithm. By transforming each attribute in a resource, LIS and MAAN generate a resource vector, the size of which equals the number of attributes in the resource. As a result, they consume much less memory for vectors. The figure also shows that for resource IDs, SFC leads to the least memory consumption and MAAN leads to the most. This is because for an l-attribute resource, MAAN produces l IDs and LIS produces five IDs, while SFC produces one ID. We see that the total memory consumption follows $\mathrm{SFC} \approx \mathrm{PIRD} > OPIRD > LIS \approx MAAN$. This result confirms the drawback of SFC and PIRD/OPIRD in building a multi-dimensional space for resource vector, and the advantage of LIS without building such a multi-dimensional space.

Fig. 4(b) shows the total number of resource predicates stored in the system versus the number of resources. We see that LIS/PIRD/OPIRD generate fewer resource predicates than MAAN. LIS/PIRD/OPIRD change each resource to n hash values regardless of the resource vector size. Therefore, each resource needs n routing messages for storing and searching. MAAN hashes each attribute of a resource and stores the resource predicate in a node. For an l-attribute resource, MAAN needs l messages for storing and searching l resource predicates. Therefore, MAAN generates many resource predicates for a resource with many attributes. SFC and LIS/PIRD/OPIRD weave all attributes of a resource into one and five ID(s), respectively, and they need one and five message(s) for storing and searching a resource. Hence, SFC generates fewer resource predicates than LIS. The experiment results imply that LIS/PIRD/OPIRD need less node communication than MAAN and more node communication than SFC for resource pooling and querying. Tough SFC is cheaper in node communication, this advantage is outweighed by its high memory consumption and low fidelity of resource searches.

Fig. 4(c) plots the average, the 1st percentile, and the 99th percentile of the number of resource predicates in a node. The average is the total number of resource predicates divided by the number of nodes with resource predicates. We see that the average number of resource predicates follows $\mathrm{MAAN} > SFC > LIS$. Due to the same reason as in Fig. 4(b), MAAN generates more resource predicates than others. Our experiment shows that SFC generates 258 different IDs, while LIS generates 2,048 different IDs. Therefore, more balanced load distribution among nodes makes LIS have lower average number than SFC. We also see that MAAN exhibits the largest



(a) Different number of nodes (b) Different number of resources

Fig. 5. Number of resource predicates.

variance, and LIS exhibits the least variance. MAAN maps resource predicates to a DHT overlay based on each attribute. Some attributes appear very frequently while others are infrequently used, leading to much more resource predicates stored in some nodes while only a few stored in others. In LIS, the widespread IDs help to distribute resource predicates evenly. In addition, it creates much fewer predicates. Therefore, its variance is not as significant as MAAN's. SFC incurs higher variance than LIS because it only relies on 258 out of 2048 nodes for all resource predicates. PIRD and OPIRD rely on a multi-dimensional space with a large number of dimensions to generate resource IDs, which may degrade the locality preservation of hashing and generate more of the same or similar IDs. Thus, they exhibit larger variance than LIS. We also observe that Hilbert-LIS has slightly larger variance than SHA-LIS and Alpha-LIS. This is because Hilbert-LIS depends on SFC to produce resource IDs, leading to more of the same or similar IDs. These experimental results show that LIS can achieve a more balanced distribution of load due to resource predicates maintenance and resource query response.

The study in [49] indicates that some attributes such as CPU utilization and free memory change dynamically. Thus, resource owners need to update their resource predicates. We tested the *updating cost* measured by the total number of update messages when the resource attributes in the system dynamically change. In this experiment, we randomly selected 5000 nodes. In each node, the number of resource attribute changes was varied from 5 to 25 with 5 increase in each step. Fig. 4(d) shows the updating cost in different systems versus the number of resource attribute changes in each node. We see that LIS/PIRD/OPIRD generate lower updating cost than MAAN but higher updating cost than SFC. Fig. 4(d) mirrors Fig. 4(b) due to the same reasons. The experimental results indicate that LIS produces acceptable updating cost for attribute dynamics.

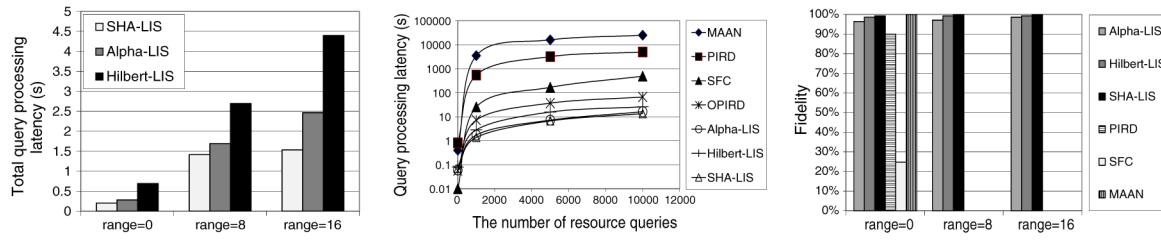
Fig. 5 shows the median, the 1st and 99th percentiles of the number of resource predicates in a node with different number of nodes and resources, respectively. Fig. 5(a) shows that the number of resource predicates in a node decreases as the number of nodes increases. As more nodes are deployed in Chord, the same load of storing resource predicates is distributed among more nodes, thus leading to less load on each node. Fig. 5(b) shows that the number of resource predicates in a node increases as the number of resources increases. More resource predicates distributed among the same number of nodes lead to more resource predicates in each node.

4.2 Efficiency and Fidelity

Recall that a range R can be used to query resources from nodes with ID $\in [ID - R, ID + R]$ to avoid missing similar resources. Fig. 6(a) plots the query processing latency of SHA-LIS, Alpha-LIS and Hilbert-LIS when R equals 0, 8, and 16 respectively. The query processing latency of a query is measured from the time when the query results are received by a requester to the time when the filtering completes. Query processing latency reflects how fast requested resources can be discovered, which affects the application performance. We see that the latency increases as the range increases. A larger range results in more located resources, which leads to a longer time for filtering unsatisfying resources. We also see that the latency follows $\mathrm{SHA-LIS} < \mathrm{Alpha-LIS} < Hilbert - LIS$. The alphanumeric mapping in Alpha-LIS takes a longer time than the consistent hashing in SHA-LIS; thus, Alpha-LIS leads to longer query latencies. Since Hilbert-LIS has one more step of Hilbert hashing than Alpha-LIS, it has longer latencies than Alpha-LIS.

Fig. 6(b) depicts the query processing latency of different services. It shows that MAAN leads to dramatically higher query processing latency than others. For a query consisting of l attributes, MAAN generates l queries and searches all resources owning each attribute. Therefore, MAAN needs a very long time to prune unsatisfying resources in the merging phase. PIRD changes each resource to n IDs, so it does not need a long time to prune unsatisfying resources and generates less latency than MAAN. By compressing resource vectors, OPIRD greatly reduces the latency of PIRD. SFC only produces one ID for each resource, leading to less latency than PIRD. However, SFC needs a longer time for processing long resource vectors than OPIRD. In LIS, the resource vector size is equal to the number of attributes in the resource description, so it produces the lowest query processing latency among all methods. This result shows the benefits of avoiding building a multi-dimensional space to produce resource IDs, and demonstrates that LIS outperforms all others in terms of query processing latency.

Recall that *fidelity* is defined as the total number of satisfying resources located divided by the total number of satisfying resources in the system. In the experiment, we use *satisfying resources* of a resource query to represent all resources in the system having more than 50% similarity with the query. We can find all satisfying resources for a query in the system offline



(a) Query processing latency of LIS (b) Query processing latency of different services (c) Fidelity

Fig. 6. Efficiency and fidelity of different resource information services.

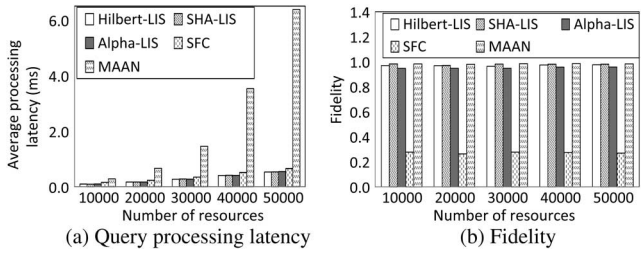


Fig. 7. Performance with a large number of resources in the system.

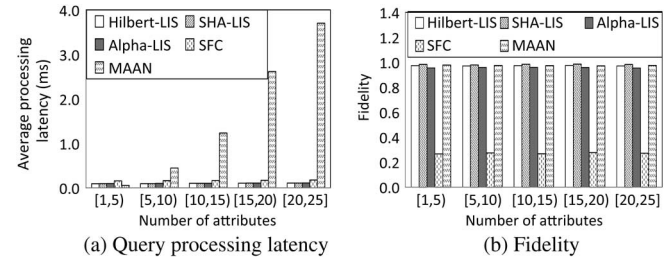


Fig. 8. Performance with variable number of attributes in a resources query.

to calculate fidelity. Fig. 6(c) depicts the fidelity of each system. Since the range querying results of PIRD, SFC, and MAAN are not comparable to the range querying results of LIS, we omit these results. Additionally, PIRD exhibits higher fidelity than OPIRD, so we omit the results of OPIRD. From the figure, we see that MAAN produces the highest fidelity, while SFC generates the lowest fidelity. Since MAAN finds all resources that have each attribute in a request before merging, it does not miss any satisfying resource in the system. SFC's effectiveness at locality preserving is degraded in a high dimensional space, so it misses around 40% of the satisfying resources. In LIS, similar resources are hashed to the same node with probability of $\geq 1-(1-p^m)^n$, so it misses about 7% of the satisfying resources. MAAN achieves high fidelity at the cost of dramatically high overhead and high query latency. LIS achieves relatively high fidelity at a significantly low overhead and high efficiency. PIRD's fidelity is not as high as those of LIS and MAAN due to the curse of dimensionality. We also observe that the fidelity follows $\mathrm{SHA-LIS} > \mathrm{Hilbert-LIS} > Alpha-LIS$. This is due to their different algorithms to transform resource attributes to integers. As the search range increases, more satisfying resources are located, leading to higher fidelity.

We then test the performance of the five methods when the system has a large number of resources and the resource queries contain variable number of attributes. Fig. 7(a) shows the average query processing latency per resource query when the number of resources was varied from 10,000 to 50,000 with 10,000 increase in each step. In this experiment, the number of attributes in each query was randomly chosen from the range of [5, 15] and the number of nodes was set to 32,768. We see as the number of resources increases, all three LIS methods and SFC produce similar query processing latency and they exhibit a small increase in query processing latency, while MAAN exhibits much higher query processing latency and dramatically higher increase rate. LIS produces five IDs and SFC produces one ID regardless of the number of attributes in a resource query. However, more resources in the system lead to more returned resources for the IDs of a query, leading to slightly longer time to prune unsatisfying resources in the merging phase. In contrast, the number of IDs of a query in MAAN equals the number of attributes in the query, which is larger than those of other methods. Also, MAAN finds all resources containing each of the query attributes. As a result, MAAN produces much more returned resources for each query and it increases faster than other methods as the number of resources increases.

From Fig. 7(b), we see that the fidelity of each algorithm stays nearly constant when the number of resources increases. Fig. 7(a) and (b) confirm that LIS maintains its high efficiency and fidelity even with a large amount of resources in the system and with a variable number of attributes in a query.

Fig. 8(a) shows the average processing latency for each resource query when there are different number of attributes in a query. $[x_1, x_2]$ in the figure means that the number of attributes was randomly chosen from the range of $[x_1, x_2]$. We see that as the number of attributes in each query increases, the query processing latency of MAAN increases significantly, while those of other methods remain nearly constant. Due to the same reasons as in Fig. 7(a), when the number of attributes in each query increases, MAAN produces more returned resources for each query and needs longer time to filter the unsatisfying results, while the query latency of the other four methods do not changes much. From Fig. 8(a) and (b), we can see that the performance of LIS is not affected by the varying number of attributes in each query.

We then studied the influence of two important parameters (m and n) on the performance of LIS. Fig. 9(a) and (b) show the fidelity and query processing latency for different values of m, respectively. We see that a larger m value requires more query processing time but leads to higher fidelity. This is because with more hash functions applied on a resource predicate, similar resources have a higher probability to be hashed to the same node. We also see that the increase rate of fidelity slows down when m increase, and $m = 20$ is the optimal value that achieves high fidelity and relatively low query processing latency. Fig. 9(c) and (d) show the fidelity and query processing latency for different values of n, respectively. We see that, similar to the observations in Fig. 9(a) and (b), a larger n value requires more query processing time but leads to higher fidelity. This is because a larger n generates a larger number of IDs for a resource predicate, then similar resources have a higher probability to be hashed to the same node. The increase rate in fidelity slows down when n increases, and $n = 5$ is an optimal value that achieves high fidelity with relatively low query processing latency.

4.3 Similar Resource Searching

If a resource has more than 50% similarity with a query, we call it *satisfying resource*; otherwise, false positive (i.e., unsatisfying resource). Fig. 10(a) shows the number of located satisfying resources and unsatisfying resources. We see that PIRD locates many false positive resources. Because of long resource vectors, PIRD may locate some resources which do not have common attributes with the query. OPIRD reduces its false positives to a certain extent but still generates much more false positives than other methods. By hashing a resource to one ID, SFC generates much fewer false positives. However, as a side effect, it misses more satisfying resources.

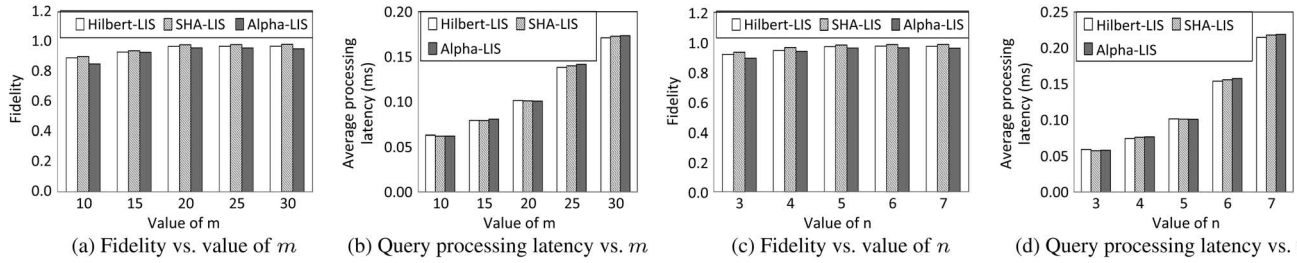


Fig. 9. Performance with different values of m and n.

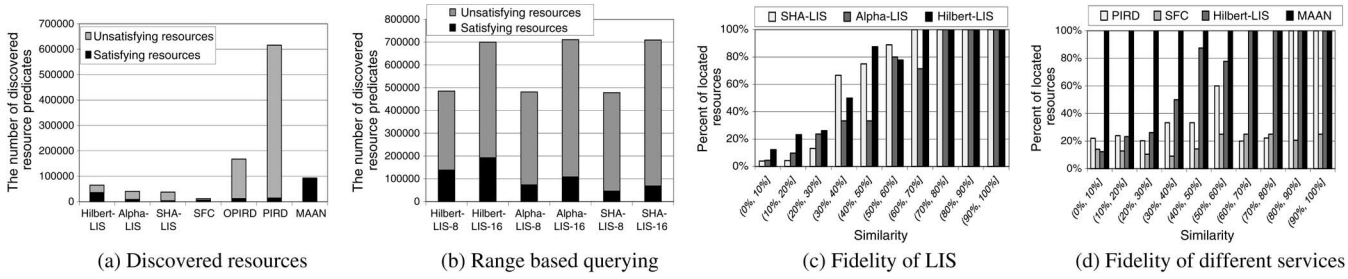


Fig. 10. Performance in similar resource searching.

SHA-LIS, Alpha-LIS, and Hilbert-LIS generate almost the same numbers of false positives, which are slightly larger than SFC's and significantly less than PIRD/OPIRD's. Hilbert-LIS can locate more satisfying resources than Alpha-LIS, which can find more satisfying resources than SHA-LIS. We also see that MAAN locates the most satisfying resources with no unsatisfying resources because it locates all resources containing each attribute in the resource query.

Fig. 10(b) shows the number of discovered resource predicates of different LIS methods with ranges equals to 8 and 16, respectively. "Hilbert-LIS-x" represents Hilbert-LIS with range x, and so on with other methods. We observe that a larger range enables the discovery of more satisfying resources, but also locates more unsatisfying resources. The total number of located resources in all algorithms is almost the same at each range, and Hilbert-LIS can find more satisfying resources than Alpha-LIS, which finds more satisfying resources than SHA-LIS. This result is consistent with that in Fig. 10(a).

The next experiment evaluates the performance of similar resource searching with $\mathrm{R} = 16$. Fig. 10(c) shows the percentage of resources located among resources with different similarities with the resource query in SHA-LIS, Alpha-LIS, and Hilbert-LIS. It shows that the percentage grows as the similarity increases. This implies that all of the algorithms can locate most resources with high similarity and filter out resources with low similarity. Each algorithm can locate all resources with similarity larger than 70%. SHA-LIS and Hilbert-LIS find all resources with similarities between 60% and 70% with the query, while Alpha-LIS discovers 71% of such resources. Thus, all LIS algorithms perform effectively in similar resource searching.

We chose Hilbert-LIS as a representative of the three LIS algorithms, and compared it with other resource information services. The results are shown in Fig. 10(d). Since PIRD performs better than OPIRD in similar resource searching, we omit the results of OPIRD here. The figure shows that MAAN always finds 100% of the resources in each group of resources with different similarities due to the same reason explained previously. Therefore, MAAN needs to filter out resources with lower similarities in the filtering process, resulting in higher overhead. In contrast, SFC locates no more than 25% of the resources in each category. The effectiveness of SFC in locality preservation decreases as the number of dimensions increases. LIS locates more than 78% of resources with similarities larger than 50%, and locates 50% of resources with similarities between 30% and 40%. Though PIRD discovers all resources with similarities between 80% and 100%, it only finds around 22% of resources with similarities between 60% and 80%. The results imply that LIS has high performance in similar resource searching by filtering out resources with low similarities and locating more resources with high similarities.

4.4 Load Balancing Algorithm

We then conducted experiments on the PlanetLab testbed in order to test the methods in the real-world distributed environment. We selected 100 PlanetLab servers across the world, placed the 2,048 nodes randomly on these servers. The capacity of nodes followed the Pareto distribution [50]–[52] with a minimum value of 25 and maximum value of 50, and a shape parameter of 1. A node with capacity c means that this node is capable of storing and handling c resource search requests. The overload degree of a node is calculated by L/C.

4.4.1 Overload Degree

Fig. 11(a), (b), and (c) show the Cumulative Distribution Function (CDF) versus node overload degree with and without the load balancing algorithm in the systems with Hilbert-LIS, SHA-LIS and Alpha-LIS, respectively. We see that the three figures show similar pattern. Without applying the load balancing algorithm, in all the three methods, about 10% of nodes suffer from overload. The load balancing algorithm guarantees that the overload degree of all nodes does not exceed 1. This result verifies the effectiveness of the load balancing in handling the resource requests. Fig. 11(d) shows the CDF versus node overload degree with and without the load balancing algorithm in SFC. We see that many nodes in SFC have 0 overload degree because it only stores resource predicates on partial nodes, as explained previously. About 12% nodes suffer from overload without the load balancing

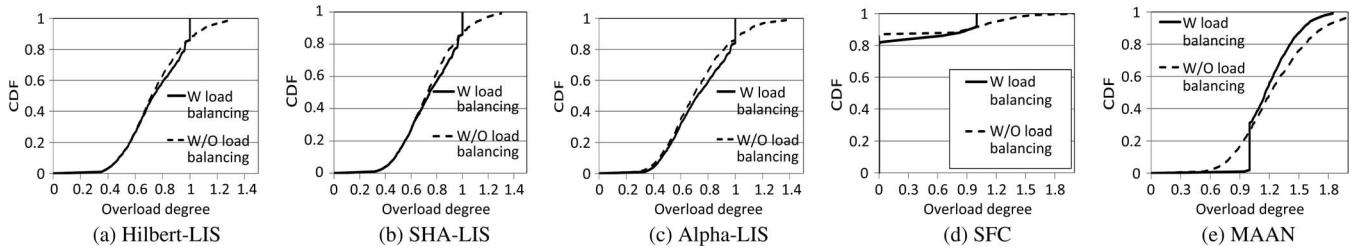


Fig. 11. Overload status with and without the load balancing algorithm.

algorithm. With the load balancing algorithm, the load is distributed more evenly and no nodes suffer from overload. Fig. 11(e) shows the overload status for MAAN.

MAAN maps resource predicates to a DHT overlay based on each attribute, leading to much more resource predicates and higher overload degree. Without load balancing, about 25% nodes have overload degree larger than 1.5, and the maximum overload degree equals 2.28. With load balancing, about 12% nodes have overload degree larger than 1.5, and the maximum overload degree equals 1.84. It is intriguing to see that even with the load balancing algorithm, MAAN still has overloaded nodes. This is because each resource has nice attributes and hence has nice IDs. In LIS, each resource has five IDs. With much more resource predicates, MAAN is heavily overloaded and some overloaded nodes cannot find lightly loaded nodes to offload their excess load.

4.4.2 Latency of Messaging

We built a 10-nary tree topology in the system for load balancing. A child reports load status to its parent by the DHT `lookup()` function. A parent reaches its child by one hop to notify it to move the load to another child. Fig. 12 shows the CDF of the number of hops that a load status message or a notification message has traveled. All messages of Alpha-LIS, Hilbert-LIS, SHA-LIS and SFC traveled no more than 15 hops, while about 10% of messages need more than 15 hops in MAAN. LIS generates five IDs and SFC generates one ID for each resource, so their generated loads are generally not heavy and available capacity for excess load can always be found within a small number of hops. MAAN generates nine IDs for each resource, so the system is overloaded and messages from overloaded nodes need to travel more hops to find available capacities. Fig. 13 plots the time needed for messages to travel through the tree. All methods approximately only need no more than 3 ms. This result indicates that our proposed load balancing algorithm only generates a very short latency.

Fig. 14 shows the total number of load transfers during the load balancing process. As previously mentioned, the workload of MAAN is the heaviest, so it needs more transfers to resolve excess workloads in the load balancing than other strategies. SFC incurs the least workload and hence the least number of transfers in load balancing.

The above experimental results show that our load balancing algorithm can balance load with limited latency and cost. We then show its effectiveness in reducing resource query latency. Fig. 15 shows the average query processing latency per resource query for all five methods. When a node is overloaded, a newly arrival request then waits in a queue until the node has available capacity. We see that the load balancing algorithm is effective in reducing the query latency; this algorithm yields more than 0.011 ms improvement for LIS, 0.013 ms improvement for SFC and 0.08 ms improvement for MAAN which suffers from heavier load.

4.5 Efficiency on the PlanetLab Testbed

We then evaluate the efficiency of the resource information services on the real-world PlanetLab testbed. We measured the following metrics. 1) Querying/storing cost, which is defined as the total number of routing hops of all messages for a query/report in a DHT overlay; 2) Querying/storing distance, which is defined as the total physical distances traveled by all messages of a query/report; 3) Querying latency, which is defined as the time period from when a requester sends out a resource query to the time when it receives query results. It includes the routing time, the query processing time at the destination and the reply time; and 4) Storing latency, which is defined as the time period needed to send a resource report to their destinations in the DHT overlay. As our three proposed algorithms produce similar performance, we select SHA-LIS as a representative of the three algorithms.

4.5.1 Querying/Storing Cost

Fig. 16(a) and (b) show the 1st percentile, median and the 99th percentile of the querying cost per query versus network size and query range R, respectively. Fig. 16(c) shows the total querying cost versus the number of queries. Fig. 16(d) plots the total storing cost with different number resources. In these four figures, the result follows $\mathrm{MAAN} > \mathrm{SHA}-\mathrm{LIS} > \mathrm{SFC}$. This is because More IDs generate more messages. MAAN generates nine query IDs for every query or resource report, larger than SHA-LIS and SFC, which generate five IDs and one ID,

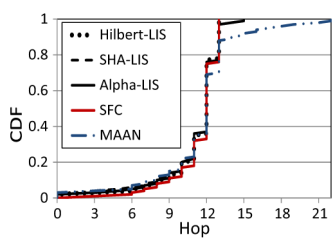


Fig. 12. The number of hops travelled by messages.

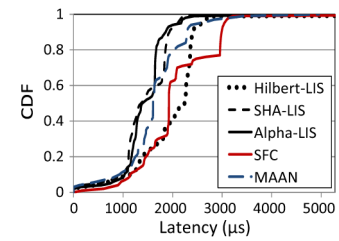


Fig. 13. Latency of messaging.

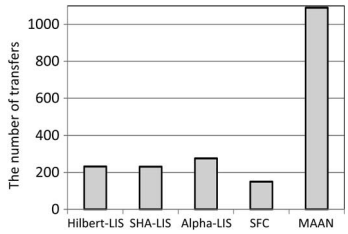


Fig. 14. Number of transfers.

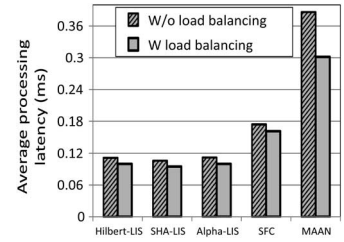


Fig. 15. Average query processing latency.

respectively. From Fig. 16(a), we see that as the network size increases, the querying cost in each service increases. This is because a query message needs to travel more hops to reach its destination in a larger-scale system. From Fig. 16(b), we see that the querying costs of all three services increase when the range is enlarged since a wider range incurs more querying hops for each query message. From Fig. 16(c), we see that the total querying cost increases as the number of queries increases because more queries lead to more hops traveled in total. Fig. 16(d) shows that more resources in the system lead to more total storing cost to store these resources.

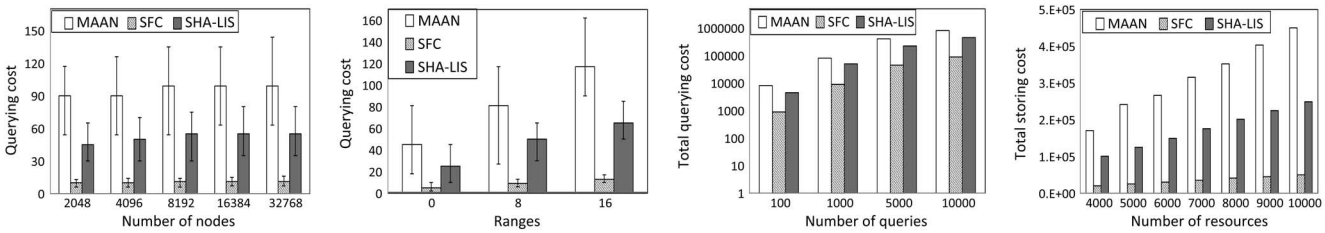
4.5.2 Querying/Storing Distance

As nodes in the system are geographically distributed in a wide area, resource querying and storing distance is critical for performance measurement.

Fig. 17(a) and (b) show the 1st percentile, median and the 99th percentile of querying distance per query versus network size and query range, respectively. Fig. 17(c) shows the total distance for different number of queries. Fig. 17(d) shows the total distance for resource storing with different number of resources. The four figures show that their results follow $\mathrm{MAAN} > \mathrm{SHA} - \mathrm{LIS} > \mathrm{SFC}$ because MAAN generates much more messages for a query or resource report than SHA-LIS, which generates more messages than SFC. Due to the same reasons explained previously, the querying distance per query increases as network size and query range grow in Fig. 17(a) and (b). Also, more queries produce higher total querying distance in Fig. 17(c). Further, as Fig. 17(d) shows, when the number of resources increases, longer total distance is needed to store resource predicates.

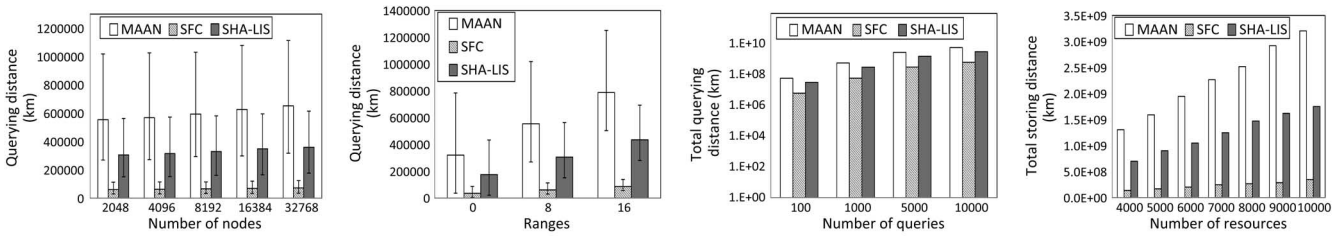
4.5.3 Querying/Storing Latency

As the query routing latency on PlanetLab is much larger than the query processing latency, so the routing latency constitutes a major part in the querying latency. Fig. 18(a) and (b) show the 1st percentile, median and the 99th percentile querying latency versus network size and query range, respectively. Fig. 18(c) shows the total querying latency for different number of queries. Fig. 18(d) shows the total storing latency with different number of resources. All these four figures show that the querying or storing latency follows $\mathrm{MAAN} > \mathrm{SHA} - \mathrm{LIS} > \mathrm{SFC}$. MAAN generates nine IDs for a query or report, SFC only generates one ID, and SHA-LIS generates five IDs. More IDs lead to a higher probability to have a higher maximum latency among the messages of the IDs. Thus, MAAN needs longer querying or storing latency than SHA-LIS, which produces longer latency than SFC. Fig. 18(a) shows that the querying latency increases as the network size grows. This is because more nodes lead to a larger size of DHT overlay hence longer routing latency. Fig. 18(b) shows that as the range value increases, the querying latency exhibits increase because the query forwarding between neighboring nodes generates latency. Fig. 18(c)



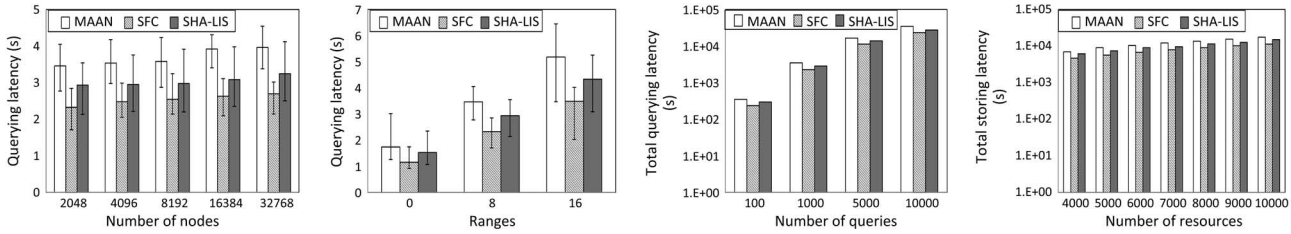
(a) Querying cost per query vs. # of nodes (b) Querying cost per query vs. range (c) Total querying cost vs. # of queries (d) Total storing cost vs. # of resources

Fig. 16. Resource querying cost and resource predicate storing cost.



(a) Querying distance per query vs. # of nodes (b) Querying distance per query vs. range (c) Total querying distance vs. # of queries (d) Total storing distance vs. # of resources

Fig. 17. Querying latency for resources and storing latency for resource predicates.



(a) Querying latency per query vs. # of nodes (b) Querying latency per query vs. range (c) Total querying latency vs. # of queries (d) Total storing latency vs. # of resources

Fig. 18. Distance for querying resource or storing resource predicates.

shows that when the number of queries increases, the total querying latency of MAAN, SFC and SHA-LIS increases. From Fig. 18(d), we see that to store 4,000 pieces of resource information, MAAN needs about 7,000 s, while SHA-LIS needs about 6,000 s, and SFC only needs about 4,600 s. Also, more resources in the system lead to higher storing distance.

All of the above results indicate that SHA-LIS generates relatively low overhead and high efficiency in resource discovery and resource information storing. Though SFC generates the least overhead, it is not comparable to SHA-LIS in terms of fidelity; MAAN generates the highest overhead and latency though it has high fidelity.

5 Conclusions

Previously proposed resource information services for large-scale resource sharing systems (e.g., collaborative cloud computing and grid computing) lead to low efficiency and high overhead or are ineffective in locating satisfying resources in an environment with a tremendous number of resource attributes. In addition, most services exhibit limited flexibility by relying on a predefined attribute list for resource description and offering only an exact-matching service. This paper presents an efficient and high-fidelity LSH based resource Information Service (LIS). LIS constructs LSH functions and relies on them to cluster the data of resources with similar attributes for efficient resource searching. More importantly, it is effective in locating satisfying resources in an environment with an enormous number of resource attributes. Furthermore, it provides high flexibility by removing the need for a predefined attribute list for resource description and similar-matching services. LIS is built on a DHT overlay, which facilitates efficient resource data pooling and searching in large-scale resource sharing systems. Extensive simulation and Planetlab experimental results demonstrate high efficiency and effectiveness of LIS in comparison to other resource information services.

SHA-LIS only offers attribute exact-matching in resource discovery, while Alpha-LIS and Hilbert-LIS provide attribute similarity search that can find attributes with similar characters (e.g., MEM and memory). In our future work, we will study the effectiveness of Alpha-LIS and Hilbert-LIS in such attribute similarity search and their side-effects caused by the alphanumeric transformation. Also, we will apply LIS to the real-world cloud environment, and develop an effective and robust resource information service application.

Acknowledgments

This research was supported in part by U.S. NSF grants IIS-1354123, CNS-1254006, CNS-1249603, CNS-1049947, CNS-0917056, CNS-1025652 and OCI-1064230, Microsoft Research Faculty Fellowship 8300751, and U.S. Department of Energy's Oak Ridge National Laboratory including the Extreme Scale Systems Center located at ORNL and DoD 4000111689. An early version of this work was presented in the Proceedings of ICCCN'09 [53].

References

[1] R. Ranjan, B. Benatallah, and M. Wang, "A cloud resource orchestration framework for simplifying the management of web applications," in *Proc. Int. Conf. Service-Oriented Comput. (ICSOC)*, 2012, pp. 248–249.

[2] J. Li, B. Li, Z. Du, and L. Meng, "Cloudvo: Building a secure virtual organization for multiple clouds collaboration," in *Proc. Int. Conf. Softw. Eng. Artif. Intell. Netw. Parallel/Distrib. Comput. (SNPD)*, 2010, pp. 181–186.

[3] C. Liu, B. T. Loo, and Y. Mao, "Declarative automated cloud resource orchestration," in *Proc. Symp. Cloud Comput. (SOCC)*, 2011, p. 26.

[4] C. Liu, Y. Mao, J. E. Van der Merwe, and M. F. Fernandez, "Cloud resource orchestration: A datacentric approach," in *Proc. Conf. Innovat. Data Syst. Res. (CIDR)*, 2011, pp. 241–248.

[5] M. Nordin, A. Abdullah, and M. Hassan, "Goal-based request cloud resource broker in medical application," *WASET*, vol. 50, no. 74, pp. 770–774, 2011.

[6] A. Goscinski and M. Brock, "Toward dynamic and attribute based publication, discovery and selection for cloud computing," in *Future Gener. Comput. Syst.*, vol. 26, no. 7, pp. 947–970, 2010.

[7] Y. Sun, T. Harmer, A. Stewart, and P. Wright, "Mapping application requirements to cloud resources," in *Proc. Int. Conf. Parallel Process. (ICPP)*, 2011, pp. 104–112.

[8] W. Yan, S. Hu, V. Muthusamy, H. Jacobsen, and L. Zha, "Efficient event-based resource discovery," in *Proc. Int. Conf. Distrib. Event-Based Syst. (DEBS)*, 2009, pp. 1–10.

[9] C. Germain, V. Neri, G. Fedak, and F. Cappello, "XtremWeb: Building an experimental platform for global computing," in *Proc. IEEE/ACM Grid*, Dec. 2000, pp. 91–101.

[10] A. Chien, B. Calder, S. Elbert, and K. Bhatia, "Entropia: Architecture and performance of an enterprise desktop grid system," *J. Parallel Distrib. Comput.*, vol. 63, no. 5, pp. 597–610, May 2003.

[11] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the grid using AppLeS," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 4, pp. 369–382, Apr. 2003.

[12] M. O. Neary, S. P. Brydon, P. Kmiec, S. Rollins, and P. Capello, "Javelin++: Scalability issues in global computing," *Future Gener. Comput. Syst. J.*, vol. 15, no. 5–6, pp. 659–674, 1999.

[13] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.

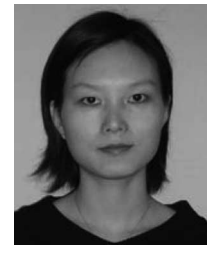
[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proc. ACM SIGCOMM*, 2001, pp. 329–350.

[15] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proc. Middleware*, 2001, pp. 329–350.

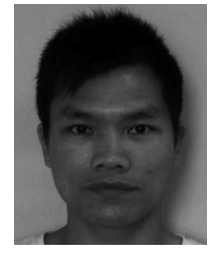
[16] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Comput. Sci. Div., Univ. of California, Berkeley, CA, Tech. Rep. UCB/CSD-01-1141, 2001.

[17] P. Maymounkov and D. Mazires, "Kademlia: A peer-to-peer information systems based on the XOR metric," in *Proc. Int. Workshop Peer-to-Peer Syst. (IPTPS)*, 2002, pp. 53–65.

[18] H. Shen, C. Xu, and G. Chen, "Cycloid: A scalable constant-degree P2P overlay network," *Perform. Eval.*, vol. 63, no. 3, pp. 195–216, 2006.
[19] M. Cai, M. Frank, and P. Szekely, "MAAN: A multi-attribute addressable network for grid information services," *Grid Comput.*, vol. 2, no. 1, pp. 3–14, 2004.
[20] M. Cai and K. Hwang, "Distributed aggregation algorithms with load-balancing for scalable grid resource monitoring," in *Proc. Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2007, pp. 1–10.
[21] A. Andrzejak and Z. Xu, "Scalable, efficient range queries for grid information services," in *Proc. Int. Conf. Peer-to-Peer (P2P) Comput.*, 2002, pp. 33–40.
[22] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting scalable multi-attribute range queries," in *Proc. ACM SIGCOMM*, 2004, pp. 353–366.
[23] H. Shen, A. Apon, and C. Xu, "LORM: Supporting low-overhead P2 P-based range-query, and multi-attribute resource management in grids," in *Proc. Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2007, pp. 1–8.
[24] C. Schmidt and M. Parashar, "Flexible information discovery in decentralized distributed systems," in *Proc. Int. Symp. High Perform. Distrib. Comput. (HPDC)*, 2003, pp. 226–235.
[25] H. Shen, Y. Zhu, Z. Li, and T. Li, "PIRD: P2P-based intelligent resource discovery in internet-based distributed systems," in *Proc. Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2008, pp. 858–865.
[26] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proc. Symp. Theory Comput. (STOC)*, 1998, pp. 604–613.
[27] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proc. Very Large Data Base (VLDB)*, 1999, pp. 518–529.
[28] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *J. Comput. Syst. Sci.*, vol. 60, no. 3, pp. 630–659, 2002.
[29] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat, "Scalable wide-area resource discovery," Comput. Sci. Div., Univ. of California, Berkeley, CA, Tech. Rep. TR CSD04-1334, 2004.
[30] S. Suri, C. Töth, and Y. Zhou, "Uncoordinated load balancing, and congestion games in P2P systems," in *Proc. Int. Workshop Peer-to-Peer Syst. (IPTPS)*, 2004, pp. 123–130.
[31] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmaier, "Space filling curves and their use in geometric data structure," *Theor. Comput. Sci.*, vol. 181, no. 1, pp. 3–15, 1997.
[32] A. Fu, P. M. S. Chan, Y. L. Cheung, and Y. S. Moon, "Dynamic VP-tree indexing for N-nearest neighbor search given pair-wise distances," *Very Large Data Base J.*, vol. 9, no. 2, pp. 154–173, 2000.
[33] T. A. Welch, "A technique for high performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, Jun. 1984.
[34] D. Karger, E. Lehman, F. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing, and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. Symp. Theory Comput. (STOC)*, 1997, pp. 654–663.
[35] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proc. Symp. Theory Comput. (STOC)*, 2002, pp. 380–388.
[36] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proc. Symp. Theory Comput. (SOTC)*, 1998, pp. 604–613.
[37] H. Shen and G. Liu, "A lightweight and cooperative multi-factor considered file replication method in structured P2P systems," *IEEE Trans. Comput.*, vol. 62, no. 11, pp. 2115–2130, 2013.
[38] H. Shen, "An efficient and adaptive decentralized file replication algorithm in P2P file sharing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 6, pp. 827–840, Jun. 2010.
[39] H. Shen, "IRM: Integrated file replication and consistency maintenance in P2P systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 1, pp. 100–113, Jan. 2010.
[40] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. DIMACS Workshop Streaming Data Anal. Mining*, 2003, pp. 253–262.
[41] H. Shen, T. Li, Z. Li, and F. Ching, "Locality sensitive hashing based searching scheme for a massive database," in *Proc. IEEE SouthEastCon*, 2008, pp. 123–128.
[42] H. Shen and G. Liu, "An efficient and trustworthy resource sharing platform for collaborative cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, 2013.
[43] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher, "Adaptive replication in peer-to-peer systems," in *Proc. Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2004, pp. 360–369.
[44] H. Shen and C.-Z. Xu, "Hash-based proximity clustering for efficient load balancing in heterogeneous DHT networks," *J. Parallel Distrib. Comput.*, vol. 68, no. 5, pp. 686–702, 2008.
[45] P. Godfrey and I. Stoica, "Heterogeneity, and load balance in distributed hash tables," in *Proc. INFOCOM*, 2005, pp. 596–606.
[46] M. Bienkowski, M. Korzeniowski, and F. M. auf der Heide, "Dynamic load balancing in distributed hash tables," in *Proc. Int. Workshop Peer-to-Peer Syst. (IPTPS)*, 2005, pp. 217–225.
[47] Y. Zhu and Y. Hu, "Efficient, proximity-aware load balancing for DHT-based P2P systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 4, pp. 349–361, Apr. 2005.
[48] S. Surana, B. Godfrey, K. Lakshminarayanan, R. Karp, and I. Stoica, "Load balancing in dynamic structured P2P systems," *Perform. Eval.*, vol. 63, no. 3, pp. 217–240, 2006.
[49] H. Bandara and A. Jayasumana, "On characteristics, and modeling of P2P resources with correlated static, and dynamic attributes," in *Proc. Global Commun. Conf. (GLOBECOM)*, 2011, pp. 1–6.
[50] K. Psounis, P. M. Fernandez, B. Prabhakar, and F. Papadopoulos, "Systems with multiple servers under heavy-tailed workloads," *Perform. Eval.*, vol. 62 (1–4), pp. 456–474, 2005.
[51] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson, "Self-similarity through high-variability: Statistical analysis of ethernet LAN traffic at the source level," *IEEE/ACM Trans. Netw.*, vol. 5, no. 1, pp. 71–86, Feb. 1997.
[52] X. Zhang, Y. Qu, and L. Xiao, "Improving distributed workload performance by sharing both CPU, and memory resources," in *Proc. Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2000, pp. 233–241.
[53] H. Shen, "Combining efficiency, fidelity, and flexibility in grid information services," in *Proc. Int. Conf. Comput. Commun. Netw. (ICCCN)*, 2009, pp. 1–6.



Haiying Shen received the BS degree in computer science and engineering from Tongji University, Shanghai, China, in 2000, and the MS and PhD degrees in computer engineering from Wayne State University, Detroit, Michigan, in 2004 and 2006, respectively. She is currently an associate professor with the Department of Electrical and Computer Engineering at Clemson University, South Carolina. Her research interests include distributed computer systems and computer networks, with an emphasis on P2P and content delivery networks, mobile computing, wireless sensor networks, and grid and cloud computing. She was the program co-chair for a number of international conferences and member of the Program Committees of many leading conferences. She is a Microsoft faculty fellow of 2010 and a member of the ACM.



Yuhua Lin received both the BS degree in software engineering and MS degree in computer science from Sun Yat-sen University, Guangdong, China, in 2009 and 2012, respectively. He is currently a Ph.D student with the Department of Electrical and Computer Engineering of Clemson University, South Carolina. His research interests include social networks and reputation systems.

Ting Li received the BS degree in computer science and computer engineering from Heilongjiang University, Harbin, China, in 2003. She also received the MS degree in software engineering from University of Bradford, U.K., in 2005. She conducted research on locality sensitive hashing when she was studying PhD in computer science and computer engineering at University of Arkansas, Fayetteville, in 2007. She is currently a programmer analyst with Wal-mart Stores Inc., Bentonville.