Journal of Grid Computing **2:** 3–14, 2004.

MAAN: A Multi-Attribute Addressable Network for Grid Information Services

Min Cai, Martin Frank, Jinbo Chen and Pedro Szekely
Information Sciences Institute, University of Southern California, Suite 1001, Marina Del Rey, CA 90292, USA
E-mail: mcai@isi.edu

Key words: Grid computing, information services, multi-attribute range queries, peer-to-peer

Abstract

Recent structured Peer-to-Peer (P2P) systems such as Distributed Hash Tables (DHTs) offer scalable key-based lookup for distributed resources. However, they cannot be simply applied to grid information services because grid resources need to be registered and searched using multiple attributes. This paper proposes a Multi-Attribute Addressable Network (MAAN) that extends Chord to support multi-attribute and range queries. MAAN addresses range queries by mapping attribute values to the Chord identifier space via uniform locality preserving hashing. It uses an iterative or single attribute dominated query routing algorithm to resolve multi-attribute based queries. Each node in MAAN only has $O(\log N)$ neighbors for N nodes. The number of routing hops to resolve a multi-attribute range query is $O(\log N + N \times s_{\min})$, where $s_{\min}$ is the minimum range selectivity on all attributes. When $s_{\min} = \varepsilon$, it is logarithmic to the number of nodes, which is scalable to a large number of nodes and attributes. We also measured the performance of our MAAN implementation and the experimental results are consistent with our theoretical analysis.

1. Introduction

Grid computing is emerging as a novel approach of employing distributed computational and storage resources to solve large-scale problems in science, engineering, and commerce. Grid computing on a large scale requires scalable and efficient resource registration and lookup. Traditional approaches maintain a centralized server or a set of hierarchically organized servers to index resource information. For example, Globus [5] uses an LDAP-based directory service named MDS [4] for resource registration and lookup. However, the centralized server(s) can become a registration bottleneck in a highly dynamic environment where many resources join, leave, and change characteristics (such as CPU load) at any time. Thus, it does not scale well to a large number of grid nodes across autonomous organizations. Also, centralized approaches have the inherent drawback of a single point of failure. Hierarchical approaches provide better scalability and failure tolerance by introducing a set of hierarchically organized servers and partitioning resource information on different servers, similar to the DNS. Typically, the partitioning scheme is predefined and can not adapt to the dynamic change of virtual organizations. Also it might take a long time for resource information to be updated from the leaf nodes to the root node.

To overcome the above shortcomings of centralized approaches, Adriana Iamnitchi et al. [8] proposed a P2P approach to organize the MDS directories in a flat, dynamic P2P network. Every virtual organization in the grid dedicates a certain amount of its resources as peers that host information services. Those peers constitute a P2P network between organizations. Resource requesters can search desired resources through query forwarding that is similar to unstructured P2P systems such as Gnutella. However, this approach does not scale well because of the large volume of query messages generated by flooding [15, 19]. In order to avoid flooding of the complete network, the number of hops on the forwarding path is typically bounded by the *Time to Live* (TTL) field of query messages. Thus, the search results are not deterministic and this approach cannot guarantee to find the desired resource even if it exists.

In contrast, recent structured P2P systems use message routing instead of flooding by leveraging a structured overlay network among peers. These systems typically support distributed hash table (DHT) functionality and the basic operation they offer is *lookup (key)*, which returns the identity of the node storing the object with the key [13]. Current proposed DHT systems include Tapestry [25], Pastry [16], Chord [21], CAN [12] and Koorde [9]. In these DHT systems, objects are associated with a key that can be produced by hashing the object name. Nodes have identifiers that share the same space as the keys. Each node is responsible for storing a range of keys and corresponding objects. The DHT nodes maintain an overlay network with each node having several other nodes as neighbors. When a *lookup (key)* request is issued from one node, the lookup message is routed through the overlay network to the node responsible for the key. Different DHT systems construct different overlay networks and employ different routing algorithms. They can guarantee to finish lookup in $O(\log N)$ or $O(dN^{1/d})$ hops and each node only maintains the information of $O(\log N)$ or d neighbors for a N nodes network (where d is the dimension of the hypercube organization of the network). Therefore, they provide very good scalability as well as failure resilience.

While DHTs have some desirable properties, they can not be directly applied to grid information services. This is because DHTs can only look up a resource that *exactly* matches the given key. Current DHT systems typically assume their applications already know the key of the target resource. For example, file systems such as CFS use DHT to index each file block and use the unique block identifier as a key to store and retrieve the block.

However, this kind of hash table functionality is not enough for grid information services because resources typically have multiple attributes and thus need to be registered with a list of attribute-value pairs. For example, a resource provider would want to register its multiple attributes like this:

```
register name = pioneer
    && url = gram://pioneer.isi.edu:8000
    && os-type = linux && cpu-speed = 1000 MHz
    && memory-size = 512 M.
```

Consequently, resource requesters want to be able to search for resources that meet multiple attribute requirements (as demonstrated by, e.g., the *Resource Specification Language* (RSL) [3] in Globus), using a query like:

```
search os-type = linux
     && 800 MHz <= cpu-speed <= 1000 MHz
     && memory-size >= 512 MB.
```

The attributes in the above example have two different types: string and numerical. Attribute “name”, “url” and “os-type” are string based and only have a limited number of values, while attribute “cpu-speed” and “memory-size” have continuous numerical values. For numerical types of attributes, being able to query with attribute ranges instead of exact values is a critical requirement. However, current DHT systems can neither handle multi-attribute queries nor range queries.

In this paper, we proposed a new structured P2P system for grid information services that we call Multi-Attribute Addressable Network (MAAN). In MAAN, resources can be registered with a set of attribute-value pairs and can be searched by multi-attribute based range queries.

The remainder of this paper introduces Chord in Section 2, describes MAAN and its routing algorithms in Section 3, presents experimental performance results of MAAN in Section 4, discusses related work in Section 5, and presents conclusions and future work in Section 6.

2. Chord

In this section, we briefly describe the Chord DHT system proposed by Ion Stoica al el. [21]. Like all other DHT systems, Chord supports scalable $\langle key, object \rangle$ pairs registration and lookup operations. Chord uses a one-dimensional circular identifier space with modulo 2^m where m is the number of bits in node identifiers and object keys. Every node in Chord is assigned a unique m-bit identifier (called the node ID) and all nodes self-organize to a ring topology based on their node IDs. The node ID can be chosen locally by hashing the node’s IP address and port number using a hashing function, such as SHA1. Each object is also assigned a unique m-bit identifier (called object key). Chord uses consistent hashing to assign keys to nodes. Key k is assigned to the first node whose identifier is

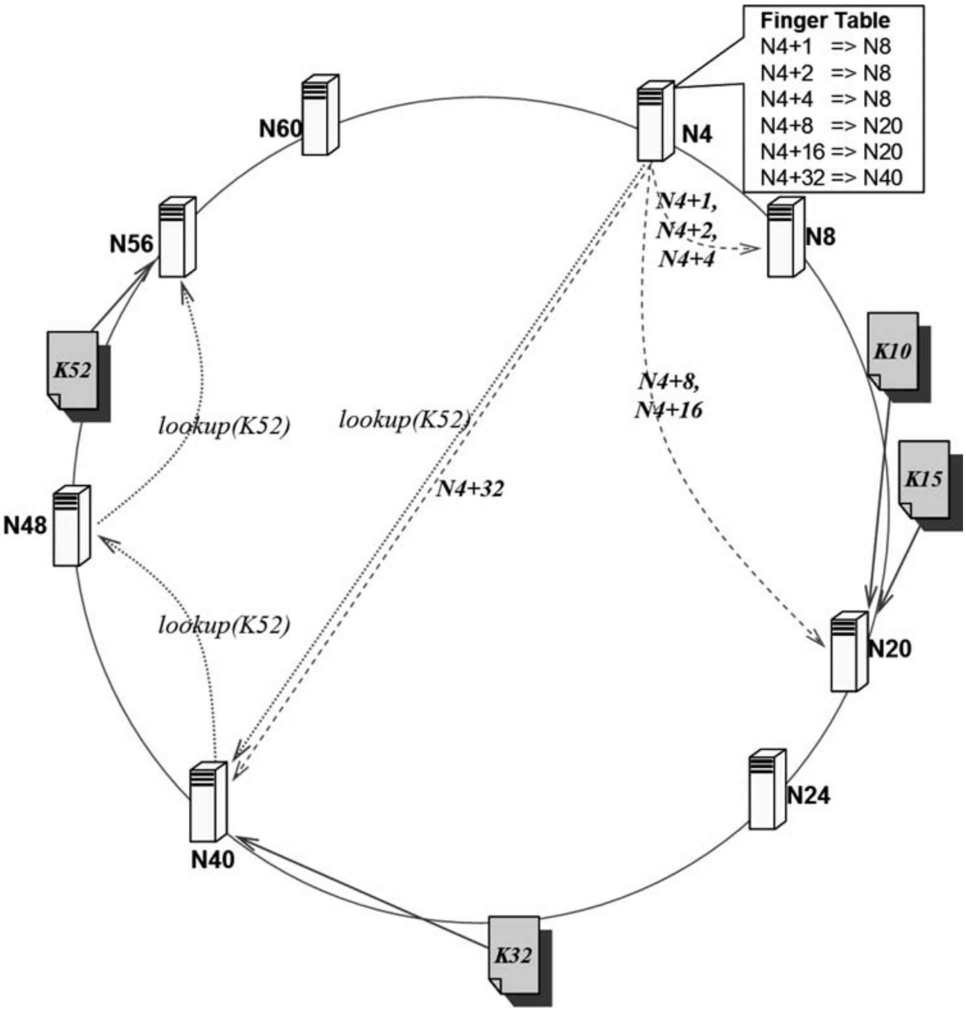


Figure 1. A 6-bit Chord network consisting of 8 nodes and 4 object keys.

equal to or follows the identifier of k in the identifier circle. This node is called the successor node of key k, denoted by $successor(k)$. Each object is registered on the successor node of its object key. Figure 1 shows an 8-node Chord network with 6-bit circular identifier space. Node $N20$ has the node ID of 20 and stores the objects with key 10 and key 15.

Each Chord node maintains two sets of neighbors, the *successor list* and *finger table*. The nodes in the successor list immediately follow the node in the identifier space, while the nodes in the finger table are spaced exponentially around the identifier space. The finger table has at most m entries. The ith entry in the table for the node with ID n contains the identity of the first node s, that succeeds n by at least 2^{i-1} on the identifier circle, i.e. $s = successor(n + 2^{i-1})$, where $1 \leqslant i \leqslant m$ (and all arithmetic is modulo 2^m). In Chord, s is called the ith finger of node n, and denoted by $n.finger[i]$. The first finger is the immediate successor of n ($i = 1$). The finger table contains more close nodes than far nodes at a doubling distance. Thus each node only needs to maintain the state for $O(\log N)$ neighbors for a Chord network with N nodes. For example, the fingers of $N4$ in Figure 1 are $N8$, $N20$ and $N40$ correspondingly.

When node n wants to search the object with key k, it will route a lookup request to the successor node x of key k, $x = successor(k)$. If node x is far away from n, n can forward the request to a far node in its finger table, which is much closer to x than n. The routing algorithm works as follows: given a lookup request with key k, the node searches its successor list for the successor of k and forwards the request to it if possible. If it does not know the successor of k, it forwards the request to the node j whose identifier most immediately precedes k in its finger list. By repeating this process, the request gets closer and closer to the successor of k. In the end, x receives the lookup request for object with key k, finds the object locally and sends the response back to n. For example, if $N4$ in Figure 1 issues a lookup request for $K52$, it sends the request to its finger $N40$ that is the closest one to $K52$ in the identifier space. $N40$ then forwards the request to $N48$ that will forward it to $N56$. Since $N56$ is the successor node of $K52$, it looks up the object corresponding to $K52$ locally and returns the result to $N4$. Because the fingers in the node's finger table are spaced exponentially around the identifier space, each hop from node n to the next node covers at least half the identifier space (clockwise) between n and k. So the average number of hops for a lookup is $O(\log N)$, where N is the number of nodes in the network.

Chord achieves the load balancing of nodes by using consistent hashing and virtual nodes. Since the node identifiers generated by SHA1 hash do not uniformly cover the identifier space, consistent hashing can not guarantee that the keys will be evenly distributed on each node. Chord solves this problem by associating keys with virtual nodes and hosting multiple virtual nodes on each real node. Chord also has a stabilization algorithm for constructing finger tables when a node joins and for maintaining finger tables when nodes fail.

However, each hop in Chord overlay might correspond to multiple hops in underlying IP network. Zhang et al. [24] proposed a lookup-parasitic random sampling (LPRS) algorithm for Chord to reduce its IP layer lookup latency. They prove that LPRS-Chord can result in lookup latencies propositional to the average unicast latency of the network, provided the underlying physical topology has power-law latency expansion.

3. Multi-Attribute Addressable Network

Like many other DHT systems, Chord offers efficient and scalable single-key based registration and lookup service for decentralized resources. However, it can not support range queries and multi-attribute based lookup. Our MAAN approach addresses this problem by extending Chord with locality preserving hashing and a recursive multi-dimensional query resolution mechanism.

3.1. *Range Queries in MAAN*

Chord assigns each node and key an m-bits identifier using a base hashing function such as SHA1, and uses consistent hash to map keys to nodes. This approach can achieve load balancing because SHA1 hash can generate randomly distributed identifiers no matter the distribution of actual node addresses and keys. However, SHA1 hashing destroys the locality of keys, and cannot support range queries for numerical attribute values.

MAAN uses SHA1 hashing to assign an m-bits identifier to each node and the attribute value with string type. However, for attributes with numerical values MAAN uses locality preserving hashing functions to assign each attribute value an identifier in the m-bit space.

DEFINITION 1. Hash function H is a locality preserving hashing function if it has the following property: $H(v_i) < H(v_j)$ iff $v_i < v_j$, and if an interval $[v_i, v_j]$ is split into $[v_i, v_k]$ and $[v_k, v_j]$, the corresponding interval $[H(v_i), H(v_j)]$ must be split into $[H(v_i), H(v_k)]$ and $[H(v_k), H(v_j)]$.

Suppose we have an attribute a with numerical values in the range of $[v_{\min}, v_{\max}]$. A simple locality preserving hashing function we can use could be $H(v) = (v - v_{\min}) \times (2^m - 1)/(v_{\max} - v_{\min})$, where $v \in [v_{\min}, v_{\max}]$. So for each attribute value v, it has the corresponding identifier $H(v)$ in the $[0, 2^m - 1]$ identifier space. MAAN also use the same consistent hashing as Chord and assign attribute value v to the successor node of its identifier, i.e. $successor(H(v))$.

THEOREM 1. *If we use locality preserving hash function H to map attribute value v to the m-bit circular space $[0, 2^m - 1]$, given a range query $[l, u]$ where l and u are the lower bound and upper bound respectively, nodes that contain attribute value v in $[l, u]$ must have an identifier equal to or larger than $successor(H(l))$ and equal to or less than $successor(H(u))$.*

Proof. Attribute value v is assigned to $successor(H(v))$ and $successor(H(v))$ is the first node whose identifier is equal to or follows the identifier of $H(v)$ in the identifier circle. Since $l \leqslant v \leqslant u$ and from Definition 1, we can see that attribute value v can only be assigned to node n and $successor(H(l)) \leqslant n \leqslant successor(H(u))$. $\square$

Thus we can use the following algorithm to resolve range queries for numeric attribute values. Suppose node n wants search for resources with attribute value v between l and u for attribute a, i.e. $l \leqslant v \leqslant u$, where l and u are the lower bound and upper bound, respectively. Node n composes a search request and uses the Chord routing algorithm to route it to node n_l, the successor of $H(l)$. The search request is as following: $\mathit{SEARCH\_REQUEST}(k, R, X)$. k is the key used for Chord routing, initially $k = H(l)$. R is the desired attribute value range: $[l, u]$ and X is a list of resources discovered in the range. Initially, X is empty. When node n_l receives the search request, it searches its local resource entries and appends those resources that satisfy the range query to X in the request. Then it checks whether it is the successor of $H(u)$ also. If true, it sends back the search response to node n with the search result in X of the search request. Otherwise, it forwards the search request to its immediate successor n_i. Node n_i also searches its local resource entries, appends matched resources to X, and forwards the request to its immediate successor until the request reaches node n_u, the successor of $H(u)$. In terms of Theorem 1, the resources that have attribute values in the range of $[l, u]$ must be registered on the nodes between n_l and n_u (clockwise) in the Chord ring. So the above search algorithm is complete. Obviously, routing the search request to node n_l using Chord routing algorithm takes $O(\log N)$ hops for N nodes. The next sequential forwarding from n_l to n_u takes $O(K)$, where K is the number of nodes between n_l and n_u. So there are total $O(\log N + K)$ routing hops to resolve a range query for single attribute. Since there are K nodes that might contain the resources matching the range query, we have to visit all of those K nodes to guarantee to find the correct search result. In this sense, $O(\log N + K)$ routing hops is optimal for range queries in Chord.

3.1.1. *Uniform Locality Preserving Hashing*

Though our simple locality preserving hashing function keeps the locality of attribute values, it does not produce uniform distribution of hashing values if the distribution of attribute values is not uniform. Consequently, the load balancing of resource entries can be poor across the nodes. To address this problem, we propose a uniform locality preserving hashing function that can always produce uniform distribution of hashing values if the distribution function of input attribute values is continuous and monotonically increasing, and is known in advance. This condition is satisfied for many common distributions, such as Gaussian, Pareto, and exponential distributions. Suppose attribute value v of resources conforms to a certain distribution with continuous and monotonically increasing distribution function $D(v)$ and possibility function $P(v) = \frac{D(v)}{\mathrm{d}v}$, and $v \in [v_{\min}, v_{\max}]$. We can design a uniform locality preserving hashing function $H(v)$ as following: $H(v) = D(v) \times (2^m - 1)$.

THEOREM 2. *Hash function $H(v)$ is a locality preserving hashing function.*

Proof. Since $D(v)$ is monotonically increasing, $H(v)$ is monotonically increasing too. Obviously, $H(v)$ is a locality preserving hashing function according to Definition 1. □

THEOREM 3. *Suppose attribute value $v \in [v_{\min}, v_{\max}]$ and v has distribution function $D(v)$. Let hashing value $y = H(v)$, then y conforms to a uniform distribution in the range of $[H(v_{\min}), H(v_{\max})]$.*

Proof. The possibility distribution of y, denoted $P(y)\,\mathrm{d}y$, is determined by the fundamental transformation law of probabilities, which is

$$|P(y)\,\mathrm{d}y| = |P(v)\,\mathrm{d}v|$$

or

$$P(y) = P(v)\left|\frac{\mathrm{d}v}{\mathrm{d}y}\right|. \tag{1}$$

Since

$$y = H(v) = D(v) \times (2^m - 1),$$

we have

$$\left|\frac{\mathrm{d}y}{\mathrm{d}v}\right| = \frac{\mathrm{d}(D(v))}{\mathrm{d}v} \times (2^m - 1)$$

or

$$\left|\frac{\mathrm{d}y}{\mathrm{d}v}\right| = P(v) \times (2^m - 1). \tag{2}$$

From (1) and (2), we have

$$P(y) = \frac{1}{2^m - 1}. \tag{3}$$

Since attribute value $v \in [v_{\min}, v_{\max}]$ and its probability function $P(v)$ is normalized by definition, as in

$$\int_{v_{\min}}^{v_{\max}} P(v)\,\mathrm{d}v = 1$$

or

$$D(v_{\max}) - D(v_{\min}) = 1.$$

Also since

$$\int_{-\infty}^{v_{\min}} P(v)\,\mathrm{d}v = 0$$

we have

$$D(v_{\min}) = 0 \quad \text{and} \quad D(v_{\max}) = 1.$$

Therefore,

$$H(v_{\min}) = D(v_{\min}) \times (2^m - 1) = 0$$

and

$$H(v_{\max}) = D(v_{\max}) \times (2^m - 1) = 2^m - 1,$$

so that

$$\int_{H(v_{\min})}^{H(v_{\max})} P(y)\,\mathrm{d}y = \int_0^{2^m-1} \frac{1}{2^m - 1}\,\mathrm{d}y = 1. \quad (4)$$

From (3) and (4), we can see that hashing value y conforms to a uniform distribution in the range of $[H(v_{\min}), H(v_{\max})]$. □

Thus, with this uniform locality preserving hashing function, resources will be uniformly distributed on all nodes if the nodes uniformly cover the m-bit identifier space. We know that the latter is true when each node hosts $O(\log N)$ virtual nodes with unrelated identifiers [21].

3.2. *Multi-Attribute Query Resolution*

Instead of only supporting one attribute based lookup, our MAAN scheme also extends the above routing algorithm for range queries to support multi-attribute lookup. In this multi-attribute setting, we assume each resource has M attributes $a_1, a_2, \ldots, a_M$ and corresponding attribute value pairs $\langle a_i, v_i \rangle$, where $1 \leqslant i \leqslant M$. For each attribute a_i, its attribute value v_i is in the range of $[v_{i\min}, v_{i\max}]$ and conforms to a certain distribution with distribution function $D_i(v)$. Thus, we can generate a uniform locality preserving hashing function $H_i(v) = D_i(v) \times (2^m - 1)$ for each attribute a_i. With these hashing functions we can map all attribute values to the same m-bit space in Chord.

Each resource will register its information (attribute value pairs) at node $n_i = \mathit{successor}(H(v_i))$ for each attribute value v_i, where $1 \leqslant i \leqslant M$. Resource registration request for attribute value v_i is routed to its successor node using Chord routing algorithm with key identifier $H(v_i)$. Each node categorizes the indices of ⟨*attribute-value, resource-info*⟩ pairs by different attributes. When a node receives a resource registration request from resource x with attribute value $a_i = v_{ix}$ and resource information r_x, it adds the $\langle v_{ix}, r_x \rangle$ pair to corresponding list for attribute a_i.

When a node searches for interested resources, it composes a multi-attribute range query that is the combination of sub-queries on each attribute dimension, i.e. $v_{il} \leqslant a_i \leqslant v_{iu}$ where $1 \leqslant i \leqslant M$, v_{il} and v_{iu} are the lower bound and upper bound of the query range, respectively.

We support two approaches to search candidate resources for multi-attribute range queries: *iterative* and *single attribute dominated* query resolution.

3.2.1. *Iterative Query Resolution*

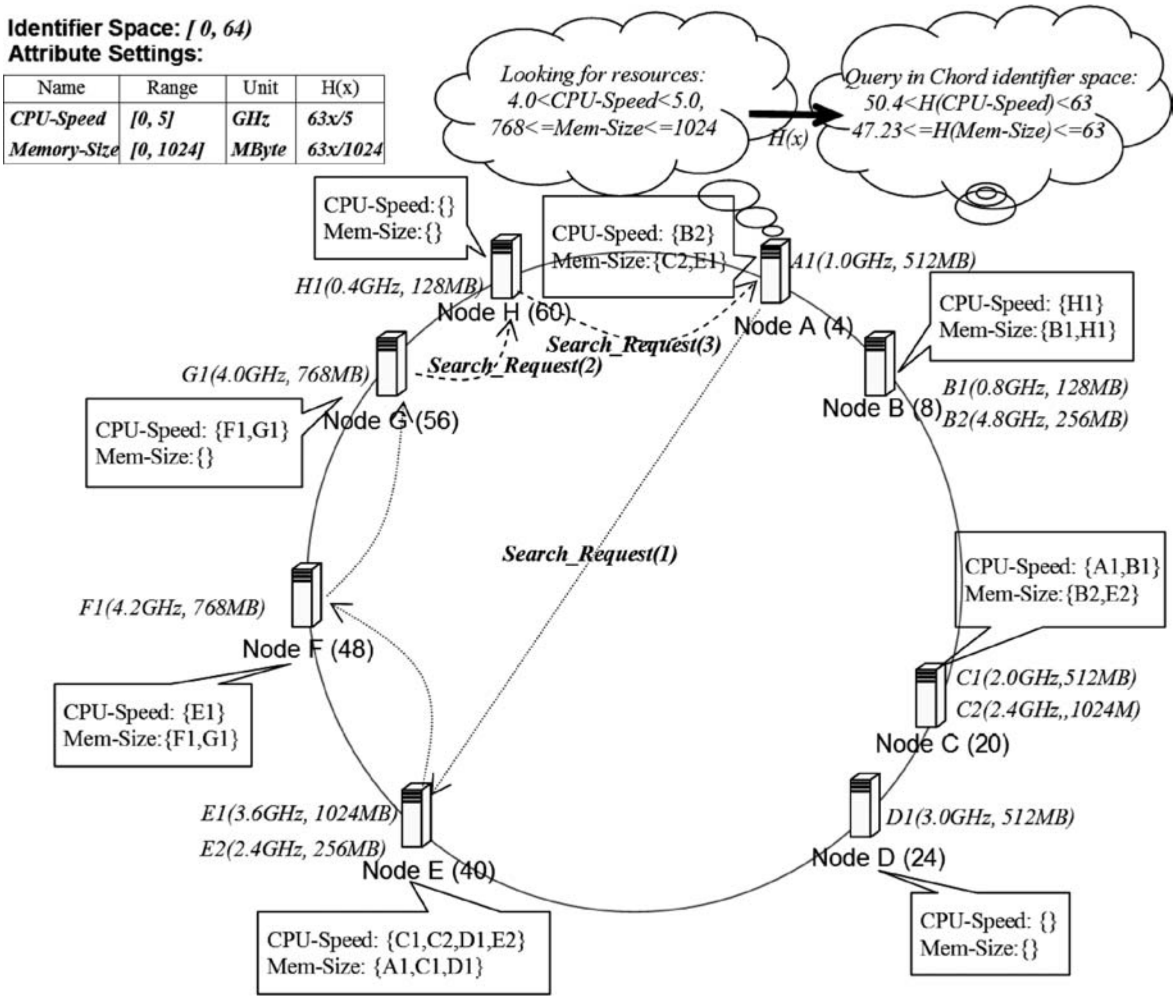
The iterative query resolution scheme is very straightforward. If node n wants to search resources by a query of M sub-queries on different attributes, it iteratively searches all candidate resources for each sub-query on one attribute dimension, and intersects these search results at query originator. We can reuse the search algorithm we proposed for single attribute based lookup in Section 3.1. The only modification is to carry a ⟨*attribute*⟩ field in each search request to indicate which attribute we are interested in. The search request is as follows: $\mathit{SEARCH\_REQUEST}(k, a, R, X)$, where a is the name of the attribute we are interested in, and k, R and X are the same as in a single attribute based query. When a node receives a query request and it intersects with the query range, it only searches the index that matches the attribute name in the search request. Though this approach is simple and easy to implement, it is not very efficient. For M-attribute queries, it takes $O(\sum_{i=1}^{M} (\log N + K_i))$ routing hops to resolve the queries, where K_i is the number of nodes intersects the query range on attribute a_i. We define selectivity s_i as the ratio of query range width in identifier space to the size of the whole identifier space, i.e.

$$s_i = \frac{H(v_{iu}) - H(v_{il})}{2^m}.$$

Suppose attribute values are uniformly distributed on all N nodes, then we have $K_i = s_i \times N$ and routing hops would be $O(\sum_{i=1}^{M} (\log N + N \times s_i))$. Thus, the routing hops for searching increase linearly with the number of attributes in the query.

3.2.2. *Single Attribute Dominated Query Resolution*

Obviously, the search result of a multi-attribute query must satisfy all the sub-queries on each attribute dimension and it is the intersection set of all resources that satisfies each individual sub-query. Suppose X is the set of resources satisfying all sub-queries, and X_i



Name	Range	Unit	H(x)
CPU-Speed	*[0, 5]*	*GHz*	*63x/5*
Memory-Size	*[0, 1024]*	*MByte*	*63x/1024*

Figure 2. An example for single-attribute-dominated query resolution algorithm.

is the set of resources satisfying the sub-query on attribute a_i, where $1 \leqslant i \leqslant M$. So we have $X = \bigcap X_i$ and each X_i is a superset of X. The iterative query resolution approach computes all X_i using M iterations and calculates their intersection set. However, since we register the resource information for each attribute dimension, resources in the set of X_i also contain the information of other attribute value pairs. The single attribute dominated query resolution approach can utilize this extra information and only need to compute a set of candidate resources X_k that satisfies the sub-query on the attribute a_k. Then it apply the sub-queries for other attributes on these candidate resources and computes the set X that satisfies all sub-queries. Here, we call attribute a_k dominated attribute. There are two possible approaches to apply these sub-queries. One approach is to apply them at the query originator after it receives all candidate resources in X_k. Since the set X_k is typically much larger than X, search requests and responses might contain many candidate resources that do not satisfy other sub-queries. Thus this approach will introduce unnecessarily large search messages and increase communication overhead. Another approach is to carry these sub-queries in the search request, and apply them locally at the nodes that contain candidate resources in X_k. This approach is more efficient because search requests and responses only carry the resources satisfying all sub-queries.

The search request in single attribute dominated approach is as follows: *SEARCH_REQUEST*(k, a, R, O, X). k, a, R are the same as those in iterative query resolve approach. O is a list of sub-queries for all other attributes except a, and X is a list of discovered resources satisfying all sub-queries. When node n wants to issue a search request with $R = [l, u]$, it first routes the request to node $n_l = \mathit{successor}(H(l))$. The node n_l searches its local index corresponding to attribute a for the resources with attribute value in the range of $[l, u]$ and with all other attributes satisfying sub-queries in O, and appends them to X. Then it checks whether it is also the successor of $H(u)$. If true, it sends back a search response to node n with the resources in X. Otherwise, it forwards the search request to its immediate successor n_s. n_s repeats this process until the search request reaches node $n_u = \mathit{successor}(H(u))$.

Since this approach only needs to do one iteration for the dominated attribute a_k, it takes $O(\log N + N \times s_k)$ routing hops to resolve the query. We can further minimize the routing hops by choosing the attribute with minimum selectivity as the dominated attribute. Thus, the routing hops will be $O(\log N + N \times s_{\min})$, where $s_{\min}$ is the minimum selectivity for all attributes.

Figure 2 shows an example of the single attribute dominated algorithm in an 8-node MAAN network storing 11 resources. This MAAN network has the identifier space of [0, 64). Each resource has two attributes: *cpu-speed* and *memory-size*. The attribute ranges and corresponding locality preserving hash functions are shown in the attribute settings table. Each node has one or more resources. For example, node B has two resources: $B1$ with 0.8 GHz CPU and 128 MB memory, and $B2$ with 4.8 GHz CPU and 256 MB memory. Each resource is registered by both cpu-speed and memory-size. For instance, resource $B1$ is registered at node C that is the successor node of its cpu-speed, and it is also registered at node B for its memory-size. When node A wants to look for resources with cpu-speed in the range of (4.0 GHz, 5.0 GHz) and memory-size in the range of [768 MB, 1024 MB], it will first apply the locality preserving hashing on each sub-query and compute the sub-queries in the Chord identifier space. It chooses the attribute with minimum selectivity as the dominated attribute, which is cpu-speed in this example. Then node A composes a search request with the hash value of lower bound as the key and routes it to the corresponding successor node G using Chord's routing algorithm. The initial search_request (1) in this example is *SEARCH_REQUEST* (50.4, *cpu-speed*, (4.0 GHz, 5.0 GHz), *memory-size* $\in$ [768 MB, 1024 MB], {*EMPTY*}). When node G receives search_ request (1), it will find the matched resource $F1$ for both sub-queries and append it into the set X. Since node G is not the successor node of upper bound, it forwards the search request to its immediate successor that is node H. The search_request (2) will be *SEARCH_REQUEST* (57, *cpu-speed*, (4.0 GHz, 5.0 GHz), *memory-size* $\in$ [768 MB, 1024 MB], $\{F1\}$). Since there is no resource registered at node H, it just simply forwards the request to node A and the search_request (3) will be the same as search_request (2) except that k is set to be 61. Node A has no matched resource for the sub-query of memory-size and it is already the successor node of the upper bound. So it just returns the resource $F1$ in set X as the search result to the search originator that happens to be itself.

Table 1. An example attribute schema for grid nodes

Attribute Name	Type	Min	Max	Unit
Name	String	/	/	/
URL	String	/	/	/
OS-Type	String	/	/	/
CPU-Speed	Numerical	1	10^5	MHz
Memory-Size	Numerical	1	10^6	MBytes
Disk-Size	Numerical	1	10^6	GBytes
Bandwidth	Numerical	10^{-3}	10^4	MBps
CPU-Count	Numerical	1	10^4	CPU

In the single attribute dominated approach, the number of routing hops is independent of the number of attributes, and thus scales perfectly in the number of attributes of a query. On the other hand, it incurs the memory cost of registering all attributes for a resource if any of its attributes is registered; and it incurs more updating overhead of attribute values change. However, the good query performance of the single attribute dominated approach will typically outweigh the greater updating cost in the Grid environment since node registration operations (of OS-Type, CPU-Speed, Memory-Size, CPU-Count, etc.) are typically far less frequent than query operations to find suitable machines.

4. Implementation and Evaluation

We verified our theoretical MAAN results by measuring the performance of an implementation in Java. It can easily be configured to support different attribute schemas, such as an example for grid nodes shown in Table 1. Our implementation runs each distributed node in its own Java virtual machine as a separated

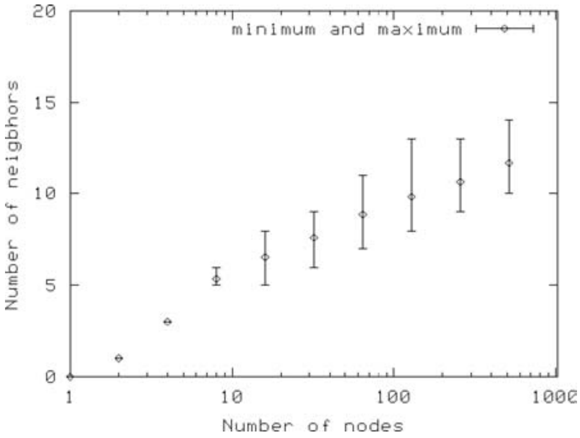
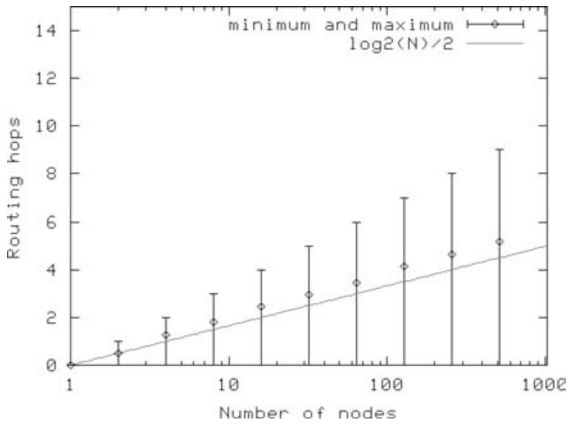


Figure 3. The number of neighbors as a logarithmic function of network size.

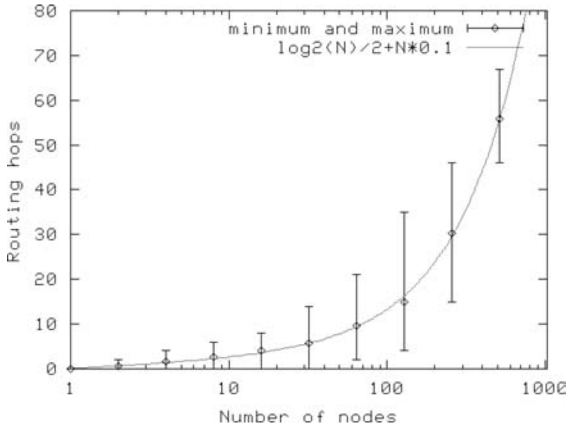
process. The implementation uses sockets to communicate between the peers, and supports the "register" and "search" commands described in the Introduction. New nodes can be added by contacting any existing peer at its IP address and port number.

To collect the performance data from the distributed nodes, we implemented a status message that is flooded to all nodes (it exists for experimental measurement purposes only). The message causes every node to dump its neighborhood state to a log file. We also instrumented MAAN messages with additional fields, such as hops taken. Our experiment environment consists of 2 dual Xeon workstations with 1 GB memory, 4 P4 desktops with 1 GB memory and 8 dual PIII workstations with 512 MB memory. The operating systems installed on these machines include Redhat 9.0, FreeBSD 4.9 and Windows XP professional. In order to setup a large MAAN network with 512 nodes, we ran up to 64 nodes each on 2 dual Xeon workstations and up to 32 nodes each on other machines. Since we use routing hops as our performance metric in this experiment, hosting multiple nodes on each machine will not affect the correctness of the results.

We know that the number of neighbors per node in Chord increases logarithmically with the network size. MAAN uses the Chord algorithm to maintain the overlay network among nodes, and thus has the same property of neighborhood states as Chord. To validate our Java implementation of MAAN, we measured the number of neighbors per node against network size. In this experiment, we set the successor replication factor to be 4, i.e. each node maintains 4 successors instead of only its immediate successor. These redundant successors will be used to recover the ring topology when nodes fail and also replicate resources. The result shown in Figure 3 confirms that similar to Chord the neighborhood states in MAAN can scale well to a large number of nodes.



(a)



(b)

Figure 4. The routing hops as a function of network size: (a) logarithmic for 5-attribute range query with $\varepsilon\%$ range selectivity; (b) linear for 2-attribute range query with 10% range selectivity.

Another important performance metric is the number of routing hops a search request would take to resolve a query. From Section 3.2, we know that the number of routing hops is $O(\log N + N \times s_{\min})$, where N is the total number of nodes in network, and $s_{\min}$ is the minimum range selectivity for all attributes. So if we want to search resources with at least one exact matching sub-query, i.e. $s_{\min} = \varepsilon\%$, the number of routing hops is $O(\log N)$, which is logarithmic to network size. Figure 4(a) shows our measurement result for 5-attribute queries with $\varepsilon\%$ range selectivity on a network with up to 512 nodes. The measured average routing hops roughly match with our theoret-

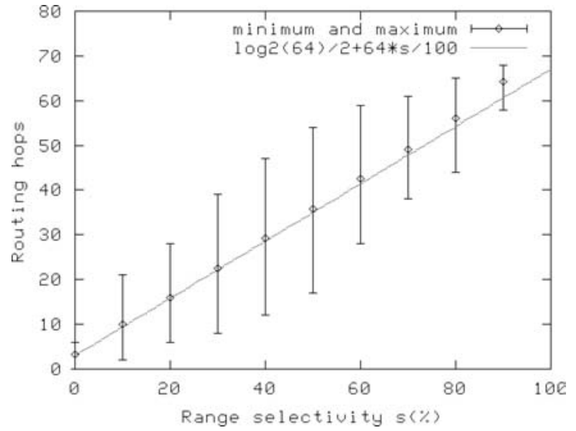


Figure 5. The routing hops as a linear function of query's range selectivity (64 nodes, 1 attribute).

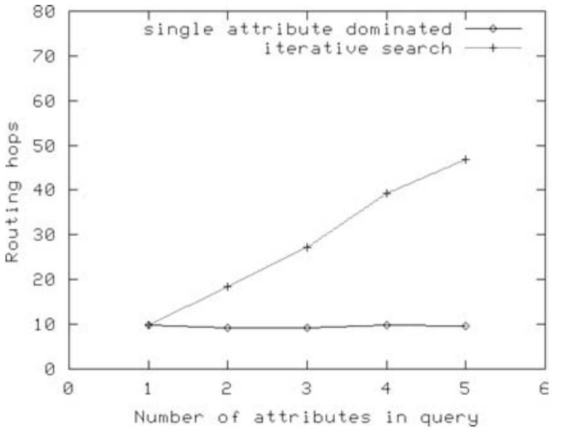


Figure 6. The expected number of routing hops as a function of the number of attributes (64 nodes, 10% range selectivity).

ical analysis as the dotted line ($\log_2(N)/2$) shows in Figure 4(a).

However, for normal range queries whose selectivity $s_i > \varepsilon\%$, the number of routing hops increases linearly with network size. This is because s_i of total N nodes have to be visited by the search queries if we want to balance the load to all the nodes. Figure 4(b) shows this linear relationship between the number of routing hops and the number of nodes for 2-attribute range queries with 10% range selectivity in a 64 nodes network. For the same reason, the number of routing hops also increases linearly with the range selectivity of search queries as shown in Figure 5. Theoretically, the average number of routing hops for range queries is $\log_2(N)/2 + N \times s_{\min}$. Our measurement result matches quite well with the analysis result, as shown by the dotted line in Figures 4(b) and 5. However we can see that range queries with large range selectivity are very costly – they will basically flood the whole network.

We also compared the two multi-attribute query resolution algorithms we proposed in Section 3.2, i.e. iterative vs. single attribute dominated. Figure 6 shows the comparison result of these two approaches. It is consistent well with our theoretical analysis.

5. Related Work

Many recent structured P2P systems are related to our research. These systems can be classified into three broad categories: *DHTs*, *tree-based*, and *skiplist-based*.

Besides Chord, other *DHT* systems include Tapestry [25], Pastry [16], CAN [12], and Koorde [9]. The routing algorithms used in Tapestry and Pastry are both inspired by Plaxton [11]. The idea of the Plaxton algorithm is to find a neighboring node that shares the longest prefix with the key in lookup message, repeat this operation until find a destination node that shares the longest possible prefix with the key. In Tapestry and Pastry, each node has $O(\log N)$ neighbors and the routing path takes at most $O(\log N)$ hops. CAN maps its keys to a d-dimensional Cartesian coordinate space that is partitioned into N zones. Each CAN node owns the zone corresponding to the mapping of its node id's on the coordinate space. The neighbors in each node are the nodes that own the contiguous zones to their local zones. Routing in CAN is straightforward: a message is always greedily forwarded to a neighbor that is closer to the key's destination in the coordinate space. Nodes in CAN have $O(d)$ neighbors and routing path length are $O(dN^{1/d})$ hops. Kaashoek and Karger [9] proved that for any constant neighborhood state k, $\Theta(\log N)$ routing hops is optimal. But in order to provide a high degree of fault tolerance, a node must maintain $O(\log N)$ neighbors. In that case, $O(\log N/\log\log N)$ optimal routing hops can be achieved. Koorde is a neighborhood state optimal DHT based on Chord and de Bruijn graphs. It embeds a de Bruijn graph on the identifier circle of Chord for forwarding lookup requests. To lookup a key k, Koorde finds the successor of k by walking down the de Bruijn graph.

TerraDir [20] is a *tree-based* structured P2P system. It organizes nodes in a hierarchical fashion according to the underlying data hierarchy. Each query request will be forwarded upwards repeatedly until reaching the node with the longest matching prefix of the query. Then the query is forward to the destination downwards the tree. In TerraDir, each node maintains constant number of neighbors and routing hops are bounded in $O(h)$, where h is the height of the tree.

Skip Graphs [2] and SkipNet [7] are two *skip-list based* structured P2P systems. Skip Graphs and SkipNet maintain $O(\log N)$ neighbors in their routing table. For each node, the neighbor at level h has the distance of 2^h to this node, i.e. they are 2^h nodes far away. This is very similar to the fingers in Chord. There are 2^h rings at level h with $n/2^h$ nodes per ring. Searching a key in Skip Graphs or SkipNet is started at the top-most level of the node seeking the key. It proceeds along the same level without overshooting the key, continuing at a lower level if required, until it reaches level 0. Their routing hops of searching a key are also $O(\log N)$.

The above structured P2P systems provide scalable distributed lookup for unique keys. However they can not support efficient search, such as keyword search and multi-dimensional range queries. Patrick Reynolds and Amin Vahdat [14] proposed an efficient distributed keyword search system, which distributes an inverted index into a distributed hash table, such as Chord or Pastry. To minimize the bandwidth consumed by multi-keyword conjunctive searches, they use bloom filters to compress the document ID sets by about one order of magnitude and use caching to exploit temporal locality in the query workload. For large sets of search results, they also use streaming transfers and return only the desired number of results.

pSearch [22] is another peer-to-peer keyword search system that distributes document indices into a CAN network based on the document semantics generated by Latent Semantic Indexing (LSI). In pSearch, the rolling index scheme is used to map the high dimensional semantic space to the low dimensional CAN space. Also it uses content-aware node bootstrapping to force the distribution of nodes in the CAN to follow the distribution of indices.

Artur Andrzejak and Z. Xu [1] extend CAN for handling range queries on single attributes by mapping one-dimensional space to CAN's multi-dimensional space using Hibert Space Filling Curve as hash function. For a range query $[l, u]$, they first route to a node whose zone includes the middle point $(l+u)/2$. Then the node recursively propagates the request to its neighbors until all the nodes that intersect the query are visited (a flooding strategy). They also proposed and compared three different flooding strategies: brute force, controlled flooding and directed controlled flooding. However, this work did not address multi-attribute range queries.

In contrast to Andrzejak's system, Cristina Schmidt and Manish Parashar [18] proposed a dimension reducing indexing scheme that efficiently maps the multi-dimensional information space into the one dimensional Chord identifier space by using Hibert Space Filling Curve. This system can support complex queries containing partial keywords, wildcards, and range queries. They solve the load balance problem by probing multiple successors at node join and migrating virtual nodes at runtime. Thus these systems do not need to know the distribution of different attribute values, but they will introduce some extra joining and migration overhead.

6. Conclusion and Future Work

In this paper, we proposed a multi-attribute addressable network (MAAN) for grid information services. It can register grid resources with a set of (*attribute,value*) pairs and search interested resources via multi-attribute based range queries. MAAN routes search queries to the nodes where the target resources are registered, and avoids flooding queries to all other irrelevant nodes.

MAAN supports efficient range queries by mapping attribute values to Chord identifier space via uniform locality preserving hashing. It not only preserves the locality of resources but also distributes resources to all nodes uniformly and achieves good load balancing among nodes. MAAN can use iterative or single attribute dominated query routing algorithm to resolve multi-attribute based queries. In MAAN, each node only maintains routing information for $O(\log N)$ other nodes. When using single attribute dominated query routing, the number of routing hops to resolve a query is $O(\log N + N \times s_{\min})$, where $s_{\min}$ is the minimum range selectivity on all attributes; thus, it scales well in the number of attributes. Also when $s_{\min} = \varepsilon$, the number of routing hops is logarithmic to the number of nodes.

While MAAN can support multi-attribute range queries quite well, it does have important limitations. First, the attribute schema of resources has to be fixed and known in advance with MAAN. We believe that supporting attribute schemas that evolve during P2P network use is an important future research direction. Second, when the range selectivity of queries is very large, flooding the query to the whole network can actually be more efficient than routing it to nodes one by one as MAAN does. It would be interesting to analyze the threshold of range selectivity at which flooding becomes more efficient than routing, and to have MAAN use different query resolution algorithms for different kind of queries.

Our current MAAN implementation uses MAAN-specific and non-standard protocol on top of TCP to communicate between nodes. However, recently the Grid community has moved to the Web Services based infrastructure, such as OGSA [23]. To be used in the real Grid environment, it is important to design and implement P2P resource information services based on standard Grid services. One approach is to implement the whole MAAN network as a distributed Grid service, which exposes a generic resource registration and discovery interface to other Grid services. In the MAAN network, each node will still uses MAAN specific protocol to communicate to each other, although they can be based on SOAP. This is similar to OpenHash [10] that provides an service-oriented DHT network instead of libraries to other applications.

Acknowledgements

We gratefully acknowledge AFOSR program funding for this project under contract number F49620-01-1-0341. We thank Ramesh Govindan for helpful discussions, and Baoshi Yan for contributing the single-attribute-dominated query resolution idea.

References

1. A. Andrzejak and Z. Xu, “Scalable, Efficient Range Queries for Grid Information Services,” in *Proceedings of the Second IEEE International Conference on Peer-to-Peer Computing*, 2002.
2. J. Aspnes and G. Shah, “Skip Graphs,” in *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003, pp. 384–393.
3. K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith and S. Tuecke, “A Resource Management Architecture for Metacomputing Systems,” Lecture Notes in Computer Science, Vol. 1495, 1998.
4. S. Fitzgerald, I. Foster, C. Kesselman, G. Laszewski, W. Smith and S. Tuecke, “A Directory Service for Configuring High-Performance Distributed Computations,” in *Proceedings of the 6th IEEE Symposium on High-Performance Distributed Computing*, 1997, pp. 365–375.
5. I. Foster and C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit,” *The International Journal of Supercomputer Applications and High Performance Computing*, Vol. 11, No. 2, pp. 115–128, 1997.
6. Gnulella. http://freenet.sourceforge.net, 2002.
7. N.J.A. Harvey, M.B. Jone, S. Saroiu, M. Theimer and A. Wolman, “SkipNet: A Scalable Overlay Network with Practical Locality Properties,” in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003.
8. A. Iamnitchi, I. Foster and D. Nurmi, “A Peer-to-Peer Approach to Resource Discovery in Grid Environments,” in *Proceedings of the 11th Symposium on High Performance Distributed Computing*, 2002.
9. F. Kaashoek and D.R. Karger, “Koorde: A Simple Degree-optimal Hash Table,” in *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS ’03)*, 2003.
10. B. Karp, S. Ratnasamy, S. Rhea and S. Shenker, “Spurring Adoption of DHTs with Open Hash, a Public DHT Servic.,” *IRP-TR-03-16*, 2003.
11. C.G. Plaxton, R. Rajaraman and A.W. Richa, “Accessing Nearby Copies of Replicated Objects in a Distributed Environment,” in *ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 311–320.
12. S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, “A Scalable Content Addressable Network,” in *Proceedings of ACM SIGCOMM*, 2001.
13. S. Ratnasamy, S. Shenker and I. Stoica, “Routing Algorithms for DHTs: Some Open Questions,” in *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS ’03)*, 2003.
14. P. Reynolds and A. Vahdat, “Efficient Peer-to-Peer Keyword Searching,” in *ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, 2003.
15. M. Ripeanu, I. Foster and A. Iamnitchi, “Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design,” *IEEE Internet Computing Journal*, Vol. 6, No. 1, 2002.
16. A. Rowstron and P. Druschel, “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems,” Lecture Notes in Computer Science, Vol. 2218, 2001.
17. S. Saroiu, P.K. Gummadi and S.D. Gribble, “A Measurement Study of Peer-to-Peer File Sharing Systems,” in *Proceedings of Multimedia Computing and Networking 2002*, 2002.
18. C. Schmidt and M. Parashar, “Flexible Information Discovery in Decentralized Distributed Systems,” in *12th IEEE International Symposium on High Performance Distributed Computing (HPDC’03)*, 2003.
19. S. Sen and J. Wong, “Analyzing Peer-to-Peer Traffic Across Large Networks,” in *Proceedings of ACM SIGCOMM Workshop on Internet Measurement Workshop*, 2002.
20. B. Silaghi, B. Bhattacharjee and P. Keleher, “Query Routing in the TerraDir Distributed Directory,” in *SPIE ITCOM’02*, 2002.
21. I. Stoica, R. Morris, D. Karger, F. Kaashoek and H. Balakrishnan, “Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications,” in *Proceedings of ACM SIGCOMM*, 2001.
22. C. Tang, Z. Xu and S. Dwarkadas, “Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks,” in *ACM SIGCOMM*, 2003.
23. S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maguire, T. Sandholm, P. Vanderbilt and D. Snelling, “The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration,” in *Global Grid Forum Draft Recommendation*, 2003.
24. H. Zhang, A. Goel and R. Govindan, “Incremental Optimization in Distributed Hash Table Systems,” *ACM Sigmetrics*, 2003.
25. B. Zhao, J. Kubiatowicz and A. Joseph, “Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing,” Technical Report UCB/CSD-01-1141, 2001.