

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

Min Cai, Martin Frank, Jinbo Chen, Pedro Szekely

Information Sciences Institute  
University of Southern California  
Marina del Rey, CA 90292  
{mcai,frank,jinbo,szekely}@isi.edu

## 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) which 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  $s_{\min} = \epsilon$ , 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 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, this centralized approach has the inherent drawback of a single point of failure. The centralized server(s) can also 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.

To overcome the above shortcomings of centralized approaches, Adriana Iamnitch 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 which host information services. Those peers constitute a P2P network between organizations. Resource requesters can search desired resources through query forwarding which 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 [13, 17]. 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 [12]. Current proposed DHT systems include Tapestry [18], Pastry [14], Chord [16], CAN [11] and Koorde [6]. In these DHT systems, objects are associated with a key which can be pro-

duced by hashing the object name. Nodes have identifiers which 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 which *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<sub>*i*</sub> 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=1000MHz
&& memory-size=512M
```

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 && 800MHz<=cpu-speed<=1000MHz
&& memory-size>=512MB
```

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 et al. [16]. 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 equal to or follows the identifier of  $k$  in the identifier circle. This node is called the successor node of key  $k$ , denoted by  $successor(k)$ . Every object is registered on the successor node of its object key.

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  $i$ -th 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  $1 \leq i \leq m$  (and all arithmetic is modulo  $2^m$ ). In Chord,  $s$  is called the  $i$ -th 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. When node  $n$  wants to search the object with key  $k$ , it will route a search 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 search 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 search request for object with key  $k$ , searches the object locally and sends the search response back to  $n$ . 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  $O(\log N)$  for a Chord system with  $N$  nodes. Also each node only needs to maintain the state for  $O(\log N)$  neighbors. Chord achieves the load balancing of nodes by using consistent hashing and virtual nodes. Chord also has a stabilization algorithm for constructing finger tables when a node joins and for maintaining finger tables when nodes fail.

### 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 uniform distribution for 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 which 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  $l \leq v \leq u$  and from Definition 1, we can see that attribute value  $v$  can only be assigned to node  $n$  and  $successor(H(l)) \leq n \leq successor(H(u))$   $\square$ .

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.  $l \leq v \leq u$ , 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: *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.

#### 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 which 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  $P(v) = \frac{D(v)}{dv}$ , 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  $\square$ .

**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  $P(y)dy$ , is determined by the fundamental transformation law of probabilities, which is

$$|P(y)dy| = |P(v)dv|$$

or

$$P(y) = P(v) \left| \frac{dv}{dy} \right| \quad (1)$$

Since

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

we have

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

or

$$\left| \frac{dy}{dv} \right| = P(v) \times (2^m - 1) \quad (2)$$

From (1) and (2), we have

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

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

$$\int_{v_{\min}}^{v_{\max}} P(v)dv = 1$$

or

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

Also since

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

we have

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

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

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

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

Thus, with this uniform locality preserving hashing function, resources will be uniformly distributed on all nodes if the node identifiers are uniformly distributed on the  $m$ -bit circular space. We know that the latter is true because we use SHA1 hashing for the node identifiers which is known to yield a uniform distribution of hash values for any input distribution [9].

### 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  $a_1, a_2, \dots, a_M$  and corresponding attribute value pairs  $\langle a_i, v_i \rangle$ , where  $1 \leq i \leq M$ . 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  $n_i = \text{successor}(H(v_i))$  for each attribute value  $v_i$ , where  $1 \leq i \leq M$ . 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  $\langle \text{attribute-value}, \text{resource-info} \rangle$  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 which is the combination of sub-queries on each attribute dimension, i.e.  $v_{il} \leq a_i \leq v_{iu}$  where  $1 \leq i \leq M$ ,  $v_{il}$  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.

#### 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: *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 which 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.

### 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 which satisfies each individual sub-query. Suppose  $X$  is the set of resources satisfying all sub-queries, and  $X_i$  is the set of resources satisfying the sub-query on attribute  $a_i$ , where  $1 \leq i \leq M$ . 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  $X_k$  which satisfies the subquery on the attribute  $a_k$ . Then it apply the sub-queries for other attributes on these candidate resources and computes the set  $X$  which 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 which 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 which contains 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 following: *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  $n_l = \text{successor}(H(l))$ . 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  $n_u = \text{successor}(H(u))$ .

Since this approach only need 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.

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 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.

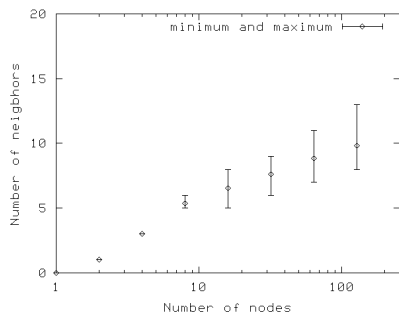
To collect the performance data from the distributed

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

**Table 1. An example attribute schema for grid nodes**

nodes, we implemented a status message which 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. We ran our experiments on a Pentium 4 2.4GHz laptop running Redhat 7.2 with 1GB memory. Since each instantiation of a peer consumes about 8MB of main memory, our experiments include up to 128 nodes.

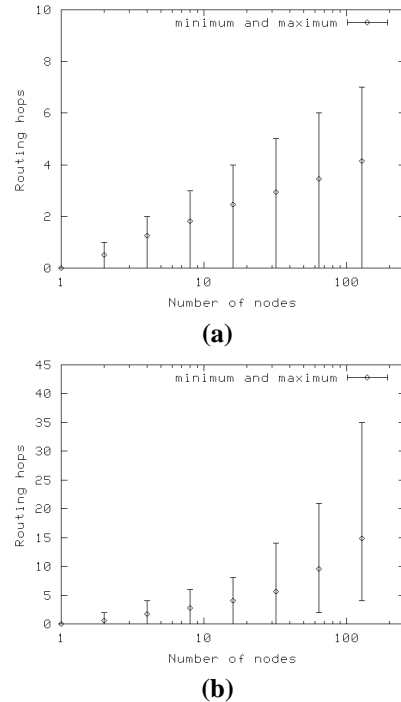
We measured the number of neighbors per node against network size. Similar to Chord, the number of neighbors at each node increases logarithmically with the network size, as shown in Figure 1. So the node state in MAAN can scale well to a large number of nodes.



**Figure 1. The number of neighbors as a logarithmic function of network size**

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,  $M$  is the number of attributes in the query 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.  $s_{\min} = \epsilon\%$ , the number of routing hops is  $O(\log N)$ , which is logarithmic to network size.

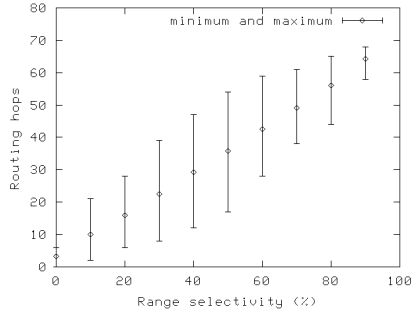
Figure 2(a) shows our measurement result for 5-attribute queries with  $\epsilon\%$  range selectivity on a network with up to 128 nodes. This result roughly matches with our theoretical analysis.



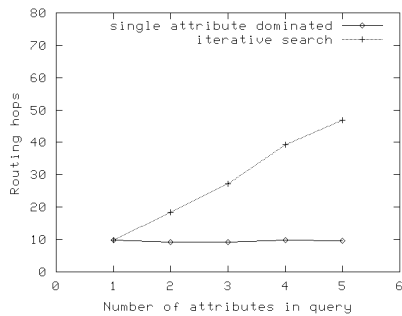
**Figure 2. The routing hops as a function of network size, (a) logarithmic for 5-attribute range query with  $\epsilon\%$  range selectivity, (b) linear for 2-attribute range query with 10% range selectivity**

However, for normal range queries whose selectivity  $s_i > \epsilon\%$ , the number of routing hops increase 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 2(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. For the same reason, the number of routing hops also increases linearly with the range selectivity of search queries. This is shown in figure 3. So 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 4 shows the comparison result of these two approaches. It is consistent well with our theoretical analysis.



**Figure 3. The routing hops as a linear function of query’s range selectivity (64 nodes, 1 attribute)**



**Figure 4. The expected number of routing hops as a function of the number of attributes (64 nodes, 10% range selectivity)**

## 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 [18], Pastry [14], CAN [11], and Koorde [6]. The routing algorithms used in Tapestry and Pastry are both inspired by Plaxton [10]. The idea of the Plaxton algorithm is to find a neighboring node which shares the longest prefix with the key in lookup message, repeat this operation until find a destination node which 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 which 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 its local zone. 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. M. F. Kaashoek et al [6] 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 request. To lookup a key  $k$ , Koorde find the successor of  $k$  by walking down the de Bruijn graph.

TerraDir [15] 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  $O(\log N)$ .

Artur Andrzejak et al [1] is most closely related to our work. It extends CAN for handling range queries on single attributes by using Space Filling Curves as hash functions. 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 which intersect the query are visited (a flooding strategy). They propose three different flooding strategies: brute force, controlled flooding and directed controlled flooding. However, the brute force strategy might visit nodes that do not actually intersect the query. In the controlled flooding strategy, nodes may receive duplicate messages for the same query. Both controlled flooding and directed controlled flooding decrease the parallelism comparing with the brute force approach. In addition, this work did not address multi-attribute range queries.

In summary, to our knowledge, no other structured peer-to-peer system has been proposed yet that addresses both multi-attribute and range queries.

## 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.

## 7. 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 (P2P2002)*, September 2002.
- [2] J. Aspnes and G. Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, Baltimore, MD, USA, 12–14 Jan. 2003.
- [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*, 1459, 1998.
- [4] S. Fitzgerald, I. Foster, C. Kesselman, G. von 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*, pages 365–375, 5-8 Aug. 1997.
- [5] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
- [6] D. R. K. Frans Kaashoek. Koorde: A simple degree-optimal hash table. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.
- [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 Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, USA, March 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*, Edinburgh, UK, August 2002.
- [9] N. I. of Standards and T. (NIST). Announcement of weakness in the secure hash standard, 1994.
- [10] 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*, pages 311–320, 1997.
- [11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.
- [12] S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for dhds: Some open questions. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.
- [13] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design, 2002.
- [14] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.
- [15] B. Silaghi, B. Bhattacharjee, and P. Keleher. Query routing in the terradir distributed directory. In *SPIE ITCOM'02*, August 2002.
- [16] 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*, 2001.
- [17] J. W. Subhabrata Sen. Analyzing peer-to-peer traffic across large networks. In *Proceedings of ACM SIGCOMM Workshop on Internet measurement workshop*, San Jose, CA, USA, November 2002.
- [18] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing, 2001.