NodeWiz: Peer-to-peer Resource Discovery for Grids

Sujoy Basu, Sujata Banerjee, Puneet Sharma, Sung-Ju Lee
Mobile and Media Systems Laboratory
HP Laboratories Palo Alto
HPL-2005-36
June 27, 2005*

E-mail: {basus,sujata,puneet,sjlee}@hpl.hp.com

Efficient resource discovery based on dynamic attributes such as CPU utilization and available bandwidth is a crucial problem in the deployment of computing grids. Existing solutions are either centralized or unable to answer advanced resource queries (e.g., range queries) efficiently. We present the design of NodeWiz, a Grid Information Service (GIS) that allows multi-attribute range queries to be performed efficiently in a distributed manner. This is obtained by aggregating the directory services of individual organizations in a peer-to-peer information service.

* Internal Accession Date Only
Published in and presented at the Fifth International Workshop on Global and Peer-to-Peer Computing, 9-12 May 2005, Cardiff, UK
Approved for External Publication

NodeWiz: Peer-to-peer Resource Discovery for Grids

Sujoy Basu, Sujata Banerjee, Puneet Sharma, Sung-Ju Lee
HP Labs, Palo Alto, CA 94304, USA
{basus,sujata,puneet,sjlee}@hpl.hp.com

Abstract

Efficient resource discovery based on dynamic attributes such as CPU utilization and available bandwidth is a crucial problem in the deployment of computing grids. Existing solutions are either centralized or unable to answer advanced resource queries (e.g., range queries) efficiently. We present the design of NodeWiz, a Grid Information Service (GIS) that allows multi-attribute range queries to be performed efficiently in a distributed manner. This is obtained by aggregating the directory services of individual organizations in a peer-to-peer information service.

1 Introduction

Efficient resource or service discovery is a crucial problem in the deployment of computing Grids, especially as these evolve to support diverse applications including interactive applications with real-time QoS requirements (e.g., multi-player networked games). As we migrate from a resource-centric world to a more service-centric one, it is anticipated that clients will search for raw computing and storage resources (e.g., machine with Pentium 1.8 GHz CPU and 512 MB memory) as well as services (e.g., Lightly loaded Everquest game service). Further, the attributes may be dynamically changing (e.g., available bandwidth between two nodes) rather than static (e.g., OS version). These trends make the resource or service discovery problem challenging. The information service must be architected to support multi-attribute range queries in an efficient manner.

Current solutions for performing these queries are either centralized or static hierarchical or have inherently poor capability for answering some complex queries. Centralized solutions do not work well in geographically large systems or with dynamic attributes that change rapidly. Many centralized solutions can be augmented by replication, but then managing consistent replicas can incur significant overhead. Hierarchical distributed systems alleviate some of the issues with the centralized systems. However, most of these are inefficient in retrieving the answers to a multi-attribute range query because of the static hierarchy through which the query has to be forwarded. Further, there is limited recourse available if due to the query load patterns, some information servers get heavily loaded while others are essentially unloaded. Other recently proposed solutions use Distributed Hash Table (DHT) technology to overcome the problems of the hierarchical systems, but do not provide a natural way to perform complex multi-attribute range queries while maintaining load-balance.

Our goal is to design a Grid Information Service (GIS) that allows multi-attribute range queries to be performed efficiently in a distributed manner. We emphasize multi-attribute range queries because these are among the more useful and common types of queries that a client would need to execute. In this paper, we present **NodeWiz** that aggregates the directory services of individual organizations in a peer-to-peer information service. NodeWiz is distributed and self-organizing such that loaded servers can dynamically offload some of their load onto other servers. Further, as described later, the information storage and organization is driven by query workloads, thereby providing a very natural way, not only to load-balance the query workload but also optimize the performance for more common multi-attribute range queries.

The next section provides the background and related work. Sections 3 describes the NodeWiz architecture in detail and presents the associated algorithms. This is followed by an evaluation using simulation in Section 4. Finally, our conclusions are presented in Section 5.

2 Background

Grid Information Service (GIS) is a key component of any large grid installation. It addresses the important problem of resource discovery which enables such large-scale, geographically-distributed, general-purpose resource sharing environments. Deployed grids based on first version of the Globus Toolkit [8] employed the Metacomputing Directory Service (MDS) [21]. The initial architecture was centralized. Subsequently, MDS-2 [5] was implemented with a

decentralized architecture. The X.500 data model used by LDAP [22] is employed in MDS-2 to organize objects in a hierarchical namespace. Each entry has one or more object classes, and must have values assigned to the mandatory attributes for these classes. Values for optional attributes may also be present. The query language, also borrowed from LDAP, allows search based on attribute values, as well as on the position of objects in the hierarchical namespace.

The MDS-2 architecture consists of directory servers and other *information providers* maintained by the different organizations participating in a grid. They use soft-state registration to join *aggregate directory* servers, which in turn can query them to get details of their content. The aggregate directory servers are expected to include generic directory servers as well as others specialized for different views and search methods. Resource brokers might want an aggregate directory that categorizes computing resources by processor speed and operating system, while monitoring applications might want an aggregate directory on running applications. Another option is to specialize based on the type of queries supported. Thus a relational aggregate directory can query the information providers for details of the resources registered by them, and enter that information in a relational database so that relational queries can be performed on them. *NodeWiz* can be viewed as a self-organizing, distributed aggregate directory that specializes in multi-attribute range queries. We envisage the information providers as the peers that come together in a peer-to-peer architecture to form NodeWiz. Hence we depart from well-known P2P networks like Napster and Gnutella most notably in the fact that we are depending on peers that are stable infrastructure nodes.

NodeWiz treats attribute values advertised by resources and services, and the queries on them in a symmetric fashion. We view the query process as distributed matchmaking in which the advertisements and queries are routed through the NodeWiz P2P network until they reach the same node where a match is found. An analogy can be made with Condor [16], which was initially designed for resource-sharing in the LAN environment, and used a centralized matchmaker. The ClassAds [20] language, used in Condor, folds the query language into the data model, by allowing resource or service descriptions, as well as queries on them, to be stated as expressions containing attribute names, relational operators, values and boolean operators. A resource provider can state in its ClassAd that a job will be accepted provided the memory required is less than 1 GB ($Memory < 1GB$). The resource consumer can state that the memory required by her job will be at most 700 MB ($Memory <= 700MB$). Matches will be found by the centralized matchmaker. Query language is not the focus of our current work. Unlike Condor, we have not addressed ranking criteria for matches found during the search. We focus on doing such multi-attribute range queries efficiently in the distributed environment.

2.1 Related Work

A prior solution for discovering resources in grid environments using a peer-to-peer approach was described in [13, 12]. Their approach differs from ours in that they use an unstructured peer-to-peer system. They do not maintain a distributed index that can efficiently lead to the nodes that can answer the query. Instead, they use heuristics such as random walks, learning-based strategy (best neighbors that answered similar query) and best-neighbor rule (one that answered most queries, irrespective of type) to contact neighbors and propagate the search through the P2P network.

INS/Twine [2] describes a peer-to-peer solution. However, the focus is on semi-structured data (e.g., in XML syntax) containing only attribute and values that may be matched. Range queries are not supported.

Distributed Hash Tables (DHT) are popular in large scale information repository systems as they are scalable, self-organizing, load balanced, and efficient. However, supporting complex queries such as range queries is difficult on DHTs. A DHT-based grid information service [1], supporting range queries on a single attribute, has studied various query request routing and update strategies. Recently, prefix hash tree [19], a trie-like data structure, has been proposed for use on top of DHT to allow range queries. PIER [11] is a distributed query engine performing database queries over a DHT. SWORD [17] is an information service that can answer multi-attribute range queries to locate suitable PlanetLab [18] nodes. SWORD sends resource advertisements to multiple sub-regions of a DHT, one per attribute. A query is routed to one of the sub-regions. MAAN [4] maintains multiple DHTs, one per attribute. Query selectivity is used to identify one attribute, and the query is routed on the corresponding DHT. Maintaining multiple overlays involves updating each of them when a resource advertisement is received. Alternately, the advertisement can be sent to one overlay, and the query must be sent to all of them. As the number of attributes increases, the update or query traffic also increases proportionately. NodeWiz maintains a single distributed index. Hence the update and query traffic is independent of the number of attributes.

There have been other proposals for supporting multi-attribute range queries in distributed environments without utilizing DHT. In [6], two spatial-database approaches are compared for supporting multi-dimensional range queries in P2P systems. The first approach uses space-filling curves to map multi-dimensional data to a single dimension. The latter is then partitioned by ranges among the available nodes. The second approach uses kd-trees to partition the

multi-dimensional space into hypercuboids, each of which is assigned to a node. In both cases, skip graphs are used to increase routing efficiency. SkipNet [10] enables range-queries on a single attribute by using the skip list data structure and ordering nodes in the overlay using string names, instead of hashed identifiers. Hence, explicit load balancing is required. Distributed Index for Multi-dimensional data (DIM) [15] is a data structure designed for multi-attribute range queries in sensor networks. It uses a geographic hash function to map the multi-dimensional space into a two-dimensional geographic space, and then uses a geographic routing algorithm. Mercury [3], like SWORD and MAAN, maintains a separate logical overlay for each attribute. Unlike them, the overlay is not a DHT. In research occurring at the same time as NodeWiz, we find Brushwood [23]. It can take any tree data structure, impose a linear ordering of the tree nodes, and add a variation of skip graphs to route efficiently. Multi-attribute range queries are supported by instantiating Brushwood with a kd-tree. It might be instructive to compare NodeWiz to some of these other approaches. We have selected SWORD, Mercury and Brushwood for this comparison. The details can be found in Appendix A.

The way we divide the attribute space among the NodeWiz nodes had some resemblance to various data structures studied in computational geometry, if we consider the attribute space as a multi-dimensional space. kd-trees divide a multi-dimensional space, but at each level of the tree, one of the dimensions is used. Interval trees organize line intervals in tree data structure, so that the intervals intersecting a query range can be efficiently found. Multidimensional range trees are recursive binary search trees. First, a balanced binary search tree is built on the first attribute, and for each subtree, all the points contained in it are used to build a balanced binary search tree on the next attribute. Since we are building a peer-to-peer distributed system, a data structure that allows efficient search in a centralized environment is not enough. We need to have efficient ways of mapping the structure among the nodes. kd-trees provide the most obvious mapping. However, using a kd-tree would imply all nodes at the same level would split on the same attribute, using the median value of the local data. In NodeWiz, nodes at the same level decide independently which attribute to split on, and the splitting value is not necessarily the median value.

3 NodeWiz Architecture

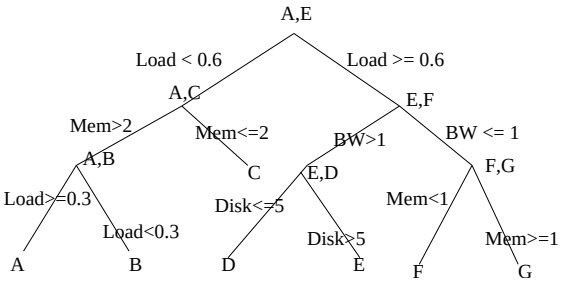
In this section, we present the Nodewiz architecture and various mechanisms for routing the queries and advertisements, and splitting the attribute subspace, etc. We will refer to the nodes in NodeWiz as the information service nodes, or service nodes interchangeably. They should not be confused with resource provider nodes on which application services will be hosted. NodeWiz adopts a soft-state approach for storing resource information for dynamic attributes. The provider nodes update the information about their service by periodically advertising the current attribute values. Resource brokers and consumer nodes will query our service nodes to find the provider nodes. They will also be referred to as clients of NodeWiz. Our emphasis in this section and the subsequent subsections is primarily on defining how service nodes join NodeWiz, how we do load balancing and how messages are routed. We do not address the issue of node failures in detail, although we mention it briefly in Section 3.3. This is because we assume that the nodes joining NodeWiz are stable infrastructure nodes, as explained in Section 2. Security issues such as mutual authentication are also outside the scope of this work.

When NodeWiz is bootstrapped with one node, the situation is similar to the centralized matchmaker in Condor [20]. When the next node joins NodeWiz, we have to distribute the workload between the new node and an existing node that was identified previously as having maximum workload. The algorithm used to identify the existing node with the highest workload is described in Section 3.1. The next step is to identify the attribute based on which the identified node will split its attribute subspace with the new node, and the splitting value of that attribute. This algorithm is described in Section 3.2.

Figure 1(a) shows how the attribute space of advertisements and queries gets eventually divided among 7 NodeWiz nodes, A through G. This figure can be viewed as a distributed decision tree according to which an advertisement or query will end up on the correct node to facilitate matchmaking. Each of the non-leaf nodes is labeled with a node-pair. The first node was an existing node of NodeWiz, while the second node joined, resulting in the split of the attribute space. Thus A was an existing node and the only node of NodeWiz when E joined. The load attribute and a splitting value of 0.6 were selected, based on the **Splitting Algorithm** described in Section 3.2, for splitting the attribute space[1]. All advertisements and queries associated with load less than 0.6 were assigned to A, while those associated with load greater than 0.6 were assigned to E. Both nodes own the same range of values for all other attributes after the split. These ranges are not affected by the split. The figure shows that E subsequently split its attribute subspace with F, and then with D. The leaves of this tree are all the existing nodes of NodeWiz.

Since the node selected for splitting is chosen with the goal of distributing the query workload evenly among all nodes, the distributed decision tree will grow in a balanced fashion, provided the query workload does not show a sud-

[1] Although we depict only binary splits, the scheme can be generalized to splitting the attribute space into more than two partitions at a given time.



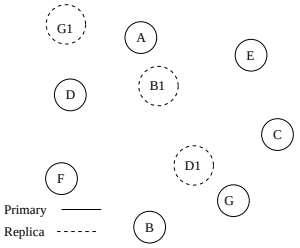
(a): Division of attribute space

Level	Attr	Min	Max	IP Addr
0	Load	0.6	+inf	E
1	Mem	0	2	C
2	Load	0	0.3	B

(b): Overlay routing table of node A

Level	Attr	Min	Max	IP Addr
0	Load	0	0.6	A
1	BW	0	1	F
2	Disk	0	5	D

(c): Overlay routing table of node E



(d): Proximity-Aware Layout of nodes

Figure 1. Division of attribute space among 7 NodeWiz nodes A through G, and corresponding routing tables of nodes A and B

den change in characteristics. In practice, the query pattern can change, and so subtrees can receive unbalanced query workloads. When the query traffic received by a node falls below a predetermined threshold,it can leave and rejoin by splitting with the currently overloaded nodes. In Figure 1(a), consider node C. If it wants to leave, it will identify the last node with which it has split the attribute subspace. In this example, it will be A. So C will inform A that it is leaving. A will remove the entry in its overlay routing table pointing to C, and also propagate the request to nodes with which it split attribute subspaces after its split with C. These nodes will repeat the action A took. This will ensure that the attribute subspace assigned to C gets reclaimed.

Each node keeps track of some of the nodes responsible for other parts of the attribute space, so that advertisements and queries can be routed efficiently. When a new node joins, it gets a copy of the routing table of the node with which it splits the latter's attribute subspace. Both nodes add an entry in their routing table pointing to the other node, and record the range of the splitting attribute assigned to the other node. We illustrate the details of the routing table in parts (b) and (c) of Figure 1, which are for nodes A and E respectively. B's table, for example, will differ from A's table only in the level 2 entry. It will have A's IP address, and the range of the load attribute assigned to A during the split with B, which is from 0.3 to 0.6. Now we consider a query for resources with $Load < 0.2$ and $Mem > 3$ which is sent to node E. Level 0 of its routing table indicates that the query should be forwarded to A. When A receives the query, and scans its routing table, there is no match at levels 0 and 1. Based on level 2, A forwards the query to B. Advertisements are routed similarly. If the query was $Load < 0.4$ and $Mem > 3$, both A and B would have received it. Thus the number of nodes visited by a query increases as it becomes less selective.

Part (d) of this figure illustrates the point that the actual location of these nodes has no correlation with the way attribute subspaces are split. For example, A and B are adjacent in the attribute space, and so queries for machines with $Load < 0.6$ and $Mem > 2$ will visit both nodes. Part (d) shows that A and B are far in terms of network latency. Hence, when a new node B1 was available for joining NodeWiz, and the workload did not warrant splitting the attribute subspace of an existing node, B1 was assigned as a replica of B. In this context, B will be referred to as the primary node. B1 was chosen as a replica of B, since A was experiencing long latency on requests forwarded to B. B1's location in the network proximity of A made it a good candidate as B's replica. B1 gets a copy of B's overlay routing table. Other NodeWiz nodes in the network proximity of B1 are alerted to its presence when they send out a query that gets routed to B. In its response, B includes B1's IP address as a replica that could be contacted. The ability to do this depends on the availability of a network distance estimation service that can order B's replicas according to their proximity to the querying node. Replicas can help localize the network traffic if many advertisements and queries generated by clients near A belong to the attribute subspace assigned to B. These clients could be served by B1. However, there is a trade-off. Depending on how consistent the replicas need to be kept with their primary nodes, there can be significant traffic generated by these additional consistency

messages. We have not evaluated the replica assignment problem so far.

3.1 Load Balancing

When a new node joins NodeWiz, we need to identify one of the most overloaded nodes and split its attribute space with the new node. We employ a distributed algorithm, henceforth referred to as the **Top-K Algorithm**, which orders the nodes in NodeWiz according to their workloads and identifies the most overloaded one. Depending on how frequently nodes join, this algorithm could run periodically or on demand. If another join request had already reached the identified node, resulting in it undergoing a split since the top-K workload information was disseminated, the request is forwarded to the node with the next highest workload identified during the run of the Top-K Algorithm.

The Top-K Algorithm is distributed and runs in two phases. Each node maintains a counter which represents its workload. The counter is incremented for each advertisement or query received by the node. Periodically, it is divided by 2 to give more weight to recent workload. In the first phase, each node sends a message to another node selected from its routing table according to a criterion to be explained soon. The recipient is selected such that these messages travel along the links of a tree composed of all the nodes in NodeWiz. Each non-leaf node waits during a timeout period for its children to send their messages to it. After receiving their messages, the node includes its own workload, sorts and retains the top K workloads along with identities of the corresponding nodes. It sends out the retained top K workloads to its selected recipient. After the root of the tree builds the list of top K workloads among all nodes, the list is disseminated in the second phase to all nodes in NodeWiz. This is simply achieved by relaying the list to all nodes from which a node receives a message in the first phase. Thus the list travels back along the links of the tree to all the children. This algorithm may run once for the on-demand case, or it may run periodically, as mentioned earlier. In the second case, there is an epoch counter tagged to each message, so that messages delayed from one epoch, do not get processed by a node in the next epoch.

We have mentioned that each node sends a message in the first phase to one of the nodes selected from its routing table. The selection of the recipient node is based on the routing table. The recipient selection process retraces the order by which nodes join NodeWiz. Recall that each join results in the splitting of the range of one attribute remaining in the possession of the splitting node. To retrace the order of these joins, each node looks at the most recent join event it participated in, either as the splitting node or as the joining node. This will be the most recent (highest level) entry in its routing table. Recall that each entry in the routing table indicates a range of values for a single attribute, and a corresponding node to which advertisements or queries overlapping that range should be sent. By excluding all ranges present in the routing table for this attribute, the node obtains the range of values of this attribute for which it is responsible. If the values in its own range are greater than the values in the range of the routing table entry, the node will wait for the recipient node in that routing table entry to send a message to it. Otherwise, the node will send its own message to the recipient node. In case the node waits for the recipient's message, it checks the next most recent entry in its routing table. This might be for the same or different attribute. In any case, a comparison is again done for the values in the range of the corresponding attribute owned by this node and the recipient of this entry. If the node has to wait for the recipient's message, it adds this recipient to the list of nodes for whose message it is waiting. This list grows until the node reaches a routing table entry, while scanning back from the most recent entry, for which the comparison indicates that it should send the message. The node does not scan the routing table beyond this point. After it waits for the messages from all the nodes its list of nodes to wait on, it includes its own workload, retains the top K values, and sends the resulting message to the recipient of the entry where it stopped scanning the routing table. Thus each node in NodeWiz will wait for zero or more nodes to send their message to it, and will send out exactly one message. The exception is the one node that will scan its entire routing table and add all nodes to its list of nodes to wait on. This is the node whose attribute subspace includes the maximum value of each attribute. This node is the root, and will disseminate the list of top K values in the second phase.

3.2 Splitting the Attribute Space

The Splitting Algorithm has to identify an attribute, for which the range of values owned by the splitting node can divided into two ranges of values. Two conditions have to be satisfied. Firstly, the values of the selected attribute in the advertisements and queries seen by the splitting node should show high probability of falling in clusters that are within the two ranges selected. This is based on the underlying assumption that an attribute which shows strong clusters will continue to do so, and has the better chance of maintaining even distribution of load between the splitting and joining nodes. For example, there might be a cluster of workstations which are kept busy by jobs submitted through a batch queuing system. There might be another cluster of desktop machines that are idle most of the time. If the splitting node finds the load averages of both sets of machines in the advertisements received by it, a clustering algorithm could easily select the load average attribute and a splitting

value so that the advertisements from the two sets of machines are assigned to the two nodes. This brings us to the second condition that needs to be satisfied. Consider the case where the clustering algorithm finds two clusters for an attribute. However one cluster is very small is size compared to the other. This can clearly lead to load imbalance between the splitting and joining node. Hence we select among all the attributes the one for which our clustering algorithm leads to most even-sized clusters. The clustering algorithm used in our experiments is the k-means algorithm [9]. The input to the algorithm in our implementation is the histogram of values of each attribute in advertisements and queries received by a node since the last time the algorithm was run. We try to divide equally the search workload of a node. When an advertisement reaches a node, pending queries are looked up, and vice-versa. Hence both queries and advertisements contribute to the histogram of each attribute.

3.3 Routing Diversity Optimization

The nodes which join NodeWiz initially, such as A and E in Figure 1(a), are found in the routing table entries of several nodes. As a result, they forward more messages than nodes that join later. We have tried a simple optimization for this problem. When a query or advertisement reaches its destination, the query results or an acknowledgment for the advertisement is sent back to the NodeWiz node that initiated the query or advertisement. When the routing diversity optimization is turned on, the initiator takes the destination node's IP address, and caches it in correlation with the routing table entry that was used to send the query or advertisement out. This ensures that another query or advertisement destined for the same sub-tree of the decision tree can be sent there with fewer overlay hops. This routing diversity optimization has been evaluated, and the results are presented in Section 4. We also observe that instead of caching just the last destination for each entry in the routing table, we can cache multiple entries. This has the added benefit of providing fault-tolerance. Also, if we allow lightly loaded nodes to leave and rejoin in a different part of the attribute space, this optimization provides an obvious way to repair the routing table.

4 Evaluation

We have built an event-driven simulation framework for NodeWiz. Our experiments use both synthetic and real datasets. For both of them, we have six attributes. In the synthetic dataset, each attribute is generated from a Pareto distribution which has been observed by other researchers to have good correlation to the attributes in a data-center trace [1]. For the real dataset, we used the measurements

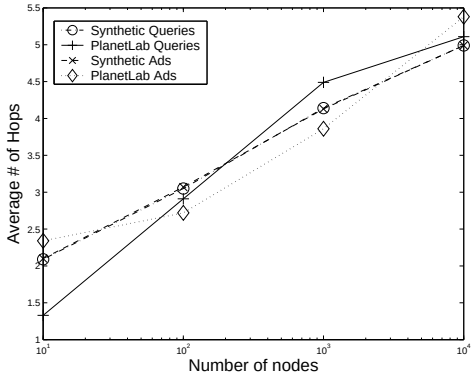


Figure 2. Average Hops for a query or advertisement as the network size increases

reported by the ganglia distributed monitoring system for PlanetLab nodes [7]. We selected six attributes from the dataset, namely the system load averages measured at 1, 5 and 15 minute intervals, and the amount of available disk, memory and swap space. Our discrete-event simulator reads one query and one advertisement at each clock cycle until all events in the input files have been consumed. The NodeWiz node which a client would contact with this query or advertisement is chosen randomly. In future, we will model network proximity to study the assignment of replicas (Figure 1(d)). At that time, the input will be specific to a client and will be sent to the nearest NodeWiz node. Each simulation must specify the number of NodeWiz nodes. When all of them have joined NodeWiz, we reset the statistics and report only the values obtained at the at the end of the simulation. The number of events simulated in the synthetic dataset is 100 times the number of nodes, and usually a third of the events are simulated by the time all nodes have joined. However, due to the small size of the PlanetLab archive available, this is not always true in the PlanetLab dataset.

Figure 2 shows the variation in average number of hops for a query or advertisement as the network size increases exponentially from 10 to 10000 nodes. We observe that the average number of hops increases very slowly. The queries in this experiment are for specific values of each attribute. If we were querying for a range, each query would visit all nodes overlapping the query range, and so the average number of hops will increase. This is explored in Figure 5, which is explained later in this section. The plots for queries and advertisements look similar. This is to be expected, since NodeWiz will treat a query and an advertisement with the same attribute values identically as long as we are not querying for a range. Both will be routed to the node with ownership of the attribute subspace in which these attribute values fall.

Figure 3 shows the increase in number of entries in the routing table, both maximum and averaged over all nodes,

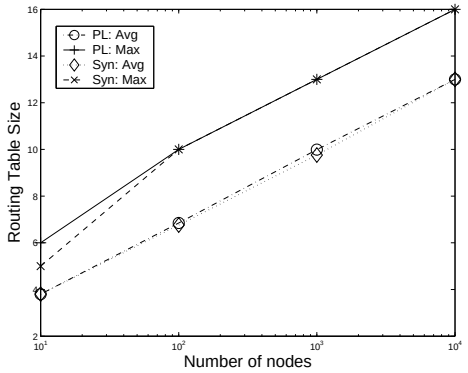


Figure 3. Maximum number of entries in the routing table as the network size increases

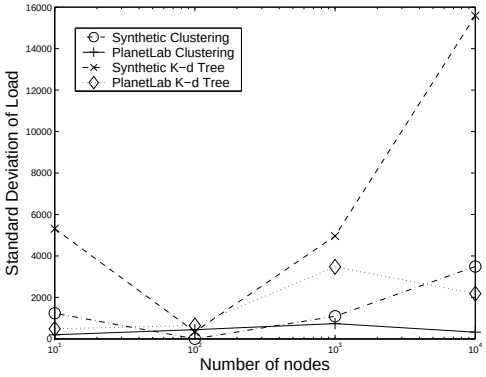


Figure 4. Standard Deviation of workload as a measure of load imbalance versus number of nodes

as the network size increases. From this figure, we conclude that the routing table size increases very slowly compared to the rate at which the network size increases. This is to be expected since the number of entries in the routing table of a node equals the number of times the attribute space has been split to obtain the node's attribute subspace. Since our joining algorithm limits the imbalance in the number of entries of different nodes, we expect this growth to be logarithmic in the number of nodes.

Figure 4 shows the standard deviation of the workload, as a measure of load imbalance, versus number of nodes. The workload is number of advertisements and queries received by a node until the end of simulation, from the steady state when all nodes have joined. Recall that we reset statistics at that point. For each dataset, we show 2 plots, one marked 'clustering' which uses the clustering algorithm described in Section 3 to identify the attribute and the splitting value. To measure how well this is doing, we compare against the plot marked 'kd-tree'. Here the idea is to divide the attribute space as a kd-tree. So, at level i in the tree, attribute i is used, with a wraparound when maximum number of attributes is reached. Also, the splitting value is the median of all data points for that attribute that the node received in advertisements. Notice that our clustering

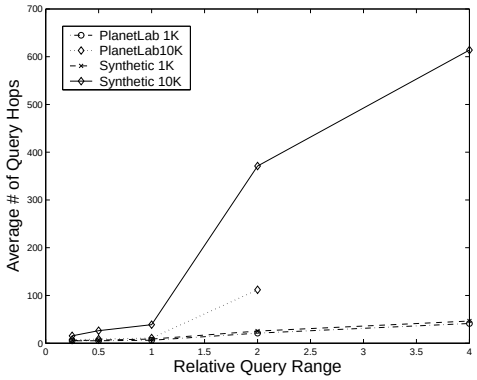


Figure 5. Average Hops for a query as the relative query range is varied. A higher value of range implies lower query selectivity

technique is doing better than a kd-tree. Also, we do better usually on PlanetLab dataset compared to the synthetic dataset. This could be attributed to the fact that the synthetic data will not have clusters as much as the real PlanetLab data. We must also note that we are not comparing to other P2P schemes that use kd-trees. In particular, the NodeWiz techniques of maintaining routing tables, and the top-K workload vector remain invariant.

Figure 5 shows that average number of hops taken by a query increases as the relative query range sought by each query increases. To obtain these range values, we computed histograms of the values of each attribute. The relative query range is 1 when the range of each attribute in the query equals the smallest inter-dectile (10 percentile) range of that attribute. Thus, larger values on the x-axis imply lower query selectivity. We observe from the figure that for the synthetic dataset, the query hops increase much faster than for the PlanetLab dataset. To understand this phenomenon, we compute the ratio of the smallest inter-dectile range to the total range of the first nine dectiles. We leave out the last one, namely values over the 90th percentile, since many of these distributions have a long tail, and very few queries overlap that region. For the synthetic dataset, this ratio is 0.016 for all attributes, since we generated them from the same Pareto distribution. For the PlanetLab dataset, this ratio was at most 0.009 for 5 attributes, and 0.02 for only 1 attribute. Clearly, for the same relative query range, a query for the synthetic dataset gets replicated more than the query on the PlanetLab dataset.

Figure 6 shows the variation in the number of attributes used by NodeWiz for the PlanetLab dataset. When we reduce the number of attributes from 6, the baseline in our experiments, to 3 and then to 1, the average number of hops taken by a query increases. This is to be expected, since each query specifies a range for each attribute. As the number of available attributes decreases, the range owned by a single node decreases for fixed number of nodes. As a

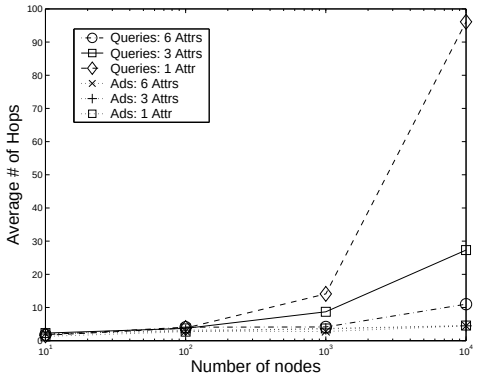


Figure 6. Query and Ad Hops versus number of nodes as the number of attributes is varied

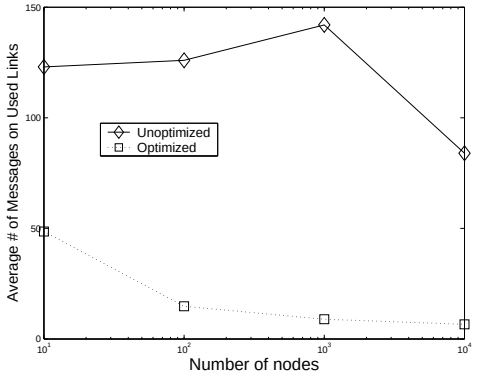


Figure 7. Effect of Routing Diversity Optimization

result, the query gets flooded to more nodes. The advertisements specify a single value, rather than a range. Hence they are insensitive to the number of attributes. Furthermore, we did not find any sensitivity of the routing table size to the number of attributes. Hence, that data has not been plotted. From Figure 6, we may also conclude that NodeWiz has an advantage over systems that support range queries using a single attribute. Unless the query has an extremely selective attribute, and a distributed index, such as a DHT, is available for that attribute, these systems will result in the query being flooded to a large number of nodes. On the other hand, our system can result in the query being flooded to a large number of nodes, only if a very large range or wildcard (any value acceptable) is specified for an attribute. This problem exists in DHT-based systems also. We can address this problem by limiting the query to some reasonable range on any attribute where a wildcard or very large range is specified.

Figure 7 shows the effect of the routing diversity optimization described in Section 3.3. We plot the average message count of used links, including both queries and advertisements, against the number of nodes. Here we are considering only overlay links on which messages were sent. As can be expected, the average number of hops decreases significantly. The reduction is by a factor of 2 for a NodeWiz deployed on 10 nodes, while it can be as much as a factor of 14 for 1000 nodes. There are two factors that contribute to this reduction. Firstly, by sending the message to the last recipient recorded in the routing table entry, we increase the probability that the message will be sent closer to its destination on the first overlay hop, as long as there is some locality in the traffic. Secondly, and more importantly, there is a significant increase in the number of overlay links that are utilized as a result of this simple optimization.

5 Conclusion

In this paper, we presented NodeWiz, a distributed and self-organizing information system for grid infrastructures. Our focus is to enable efficient execution of multi-attribute range queries, which are expected to be an important and common class of queries. NodeWiz allows for information service nodes to be dynamically added and removed from the information system to address scalability and performance concerns. More specifically, the algorithms described as part of the NodeWiz system have the capability to balance the load across multiple information service nodes while optimizing the performance for popular multi-attribute range queries in a distributed manner. The prior work on this problem does not provide a natural way to deal with these kind of queries.

In NodeWiz, advertisements from service providers are placed strategically into the information system such that queries from the service consumers are routed efficiently (with minimum number of hops) to the nodes where the matching advertisements reside. We evaluated our algorithms using simulations on synthetic and PlanetLab data. We presented results on the average number of hops for a query or advertisement as the network size (number of nodes), number of attributes and query selectivity are varied. We also evaluated load imbalance and a routing optimization. The preliminary results obtained indicate that NodeWiz has an advantage over systems that consider single attributes in isolation.

Our future work includes deploying NodeWiz on PlanetLab and obtaining real world performance data. We also intend to do a comprehensive evaluation and tuning of the proposed algorithms.

References

[1] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Proceedings of the IEEE P2P2002*, September 2002.

[2] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Proceedings of the Pervasive 2002*, August 2002.

[3] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of the ACM SIGCOMM 2004*, August 2004.

[4] M. Cai, M. Frank, J. Chen, and P. Szekely. Maan: A mulit-attribute addressable network for grid information services. In *Proceedings of the 4th International Workshop on Grid Computing (Grid2003)*, 2003.

[5] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselmann. Grid information services for distributed resource sharing. In *Proceedings of the IEEE HPDC-10*. IEEE Press, 2001.

[6] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multi-dimensional queries in p2p systems. In *Proceedings of the WebDB 2004*, June 2004.

[7] Ganglia Archives for PlanetLab. http://planetlab.millennium.berkeley.edu/.

[8] Globus Toolkit. http://www.globus.org/toolkit/.

[9] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*, chapter 8: Cluster Analysis, pages 349–351. Morgan Kaufmann Publishers, 2001.

[10] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A scalable overlay network with practical locality properties. In *Proceedings of the USITS 2003*, March 2003.

[11] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of the VLDB 2003*, 2003.

[12] A. Iamnitchi and I. Foster. On fully decentralized resource discovery in grid environments. In *Proceedings of the International Workshop on Grid Computing*, November 2001.

[13] A. Iamnitchi and I. Foster. A peer-to-peer approach to resource location in grid environments. In J. Weglarz, J. Nabrzyski, J. Schopf, and M. Stroinski, editors, *Grid Resource Management*. Kluwer Publishing Company, 2003.

[14] D. Karger and M. Ruhl. Simple effi cient load balancing algorithms for peer-to-peer systems. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43, 2004.

[15] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multidimensional range queries in sensor networks. In *Proceedings of the ACM SenSys 2003*, November 2003.

[16] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988.

[17] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed resource discovery on planetlab with SWORD. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS 2004)*, December 2004.

[18] PlanetLab http://www.planet-lab.org/.

[19] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief announcement: Prefi x hash tree. In *Proceedings of the ACM PODC 2004*, July 2004.

[20] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the IEEE HPDC-7*, Chicago, IL, 1998.

[21] G. von Laszewski, S. Fitzgerald, I. Foster, C. Kesselman, W. Smith, and S. Tuecke. A Directory Service for Confi guring High-Performance Distributed Computations. In *Proceedings of the IEEE HPDC-6*, pages 365–375, 1997.

[22] M. Wahl, T. Howes, and S. Kille. Rfc 2251: Lightweight directory access protocol (v3).

[23] C. Zhang, A. Krishnamurthy, and R. Wang. Brushwood: Distributed trees in peer-to-peer systems. In *Proceedings of the 4th International Workshop on Peer-To-Peer Systems (IPTPS '05)*, 2005.

A Comparison of NodeWiz with Other Approaches

Table 1 shows a comparison of NodeWiz with 3 other systems which we have briefly described in Section 2.1. These are SWORD [17], Mercury [3] and Brushwood [23]. We compare them on 4 points. The first one is "Index distribution", which refers to how the multi-attribute space is divided among the nodes of the system to create a distributed index. In SWORD, each attribute is assigned to a different sub-region of a common DHT. This is done by the choice of the DHT's hash function. It segments the keyspace by encoding the attribute in the higher-order bits. Mercury's overlays are called routing hubs. There is one per attribute logically, so that each node can potentially participate in multiple overlay. Each node in a logical overlay is responsible for a range of values of that attribute. It maintains neighbor links, and also long-distance links that are adapted according to load distribution. In NodeWiz, the attribute space is divided according to a tree data structure, and each leaf node of this tree is assigned to a node of the NodeWiz system. Each NodeWiz node maintains an overlay routing table for efficient routing. Brushwood has an architecture in which nodes of any tree data structure can be subjected to a linear ordering, and skip-graph pointers are added for efficient routing. For multi-attribute range queries, Brushwood is instantiated with kd-tree. The next two columns of the table compare these systems on how advertisements and range queries are routed. Both SWORD and Mercury make the design choice that advertisements will be replicated for each attribute that is indexed. This allows queries to be routed to only one set of nodes, namely those responsible for the most selective attribute. The alternative is to index the advertisements on only one attribute, and send each query to all the overlays. NodeWiz and Brushwood do not need to make this tradeoff. They can treat queries and advertisements symmetrically, sending them only to the nodes intersecting the relevant parts of the attribute space in the query. NodeWiz accomplishes this using its overlay routing tables, while Brushwood utilizes its skip-graph pointers. The final column of the table compares the systems based on their load-balancing approach. The randomization obtained from a DHT's hash function, such as in SWORD, is not enough since skewed distribution of advertisements or queries will overload a small set of nodes. To address this problem, adapting to the dynamic distribution is necessary.

Name	Index Distribution	Advertisement Routing	Range Query Routing	Load Balancing
Sword	Each attribute is assigned a different sub-region of a common DHT.	Publish in every sub-region of the DHT.	Send to the sub-region of a random attribute or the most selective attribute	Leave-rejoin protocol (Karger and Ruhl's algorithm) combined with customized hash functions
Mercury	One Routing Hub for each Attribute. Long-distance links adapted according to load distribution	Publish in every Hub.	Forward to the most selective attribute's routing hub	Leave-rejoin protocol based on sampled histogram of load around the routing hub
NodeWiz	Attribute space is divided and distributed based on workload, and overlay routing table is added at each node	Follow the overlay routing table to find the responsible nodes.	Same as Advertisement	Workload-based division of attribute space during join, based on propagated load information. Leave-rejoin is also possible.
Brushwood	Tree nodes are distributed based on linear ordering, and skip-graph pointers are added	Follow the skip-graph pointers to find the responsible nodes	Same as Advertisement	New joins and and leave-rejoin protocol based on propagated load information.

Table 1. In this table, the Index Distribution column indicates how the multi-attribute space is divided to create a distributed index

SWORD employs a leave-rejoin protocol, based on Karger and Ruhl's algorithm [14]. Mercury and Brushwood also employ a leave-rejoin protocol, but depend on propagated load information rather than random key generation as in SWORD. NodeWiz also depends on propagated load information, and adaptation to the load has been studied only in the context of new nodes joining. The leave-rejoin protocol is part of our future work.