Scalable Wide-Area Resource Discovery

David Oppenheimer[†], Jeannie Albrecht[‡], David Patterson[†], and Amin Vahdat[‡]

[†]EECS Computer Science Division
University of California Berkeley
{*davidopp,patterson*}*@cs.berkeley.edu*

[‡]Department of Computer Science and Engineering
University of California San Diego
{*jalbrecht,vahdat*}*@cs.ucsd.edu*

ABSTRACT

This paper describes the design and implementation of SWORD, a scalable resource discovery service for wide-area distributed systems. SWORD locates a set of machines matching user-specified constraints on both static and dynamic node characteristics, including both single-node and inter-node characteristics. We explore a range of system architectures to determine the appropriate tradeoffs for building a scalable, highly-available, and efficient resource discovery infrastructure. We describe: i) techniques for efficient handling of multi-attribute range queries that describe application resource requirements; ii) an integrated mechanism for scalably measuring and querying inter-node attributes without requiring $O(n^2)$ time and space; iii) a mechanism for users to encode a restricted form of utility function indicating how the system should filter candidate nodes when more are available than the user needs, and an optimizer that performs this node selection based on per-node and inter-node characteristics; and iv) working prototypes of a variety of architectural alternatives—running the gamut from centralized to fully distributed—along with a detailed performance evaluation. SWORD is currently deployed as a continuously-running service on PlanetLab. We find that SWORD offers good performance, scalability, and robustness in both an emulated environment and a real-world deployment.

1. INTRODUCTION

Large-scale distributed services such as content distribution networks, peer-to-peer storage, distributed games, and scientific applications, have recently received substantial interest from both researchers and industry. At the same time, shared distributed platforms such as PlanetLab [3] and the Grid [10, 9] have become popular environments for evaluating and deploying such services. One significant difficulty in the practical use of such shared, large-scale infrastructures centers around locating an appropriate subset of the system to host a service, computation, or experiment.

This choice may be dictated by a number of factors, depending on the application's characteristics. "Compute-intensive" applications, such as embarrassingly parallel scientific applications, might be particularly concerned about spare CPU, physical memory, and disk capacity on candidate nodes. "Network-intensive" applications, such as content distribution networks and security monitoring applications, might be particularly concerned about placing service instances at particular network locations—near potential users or at well-distributed locations in a topology—and on nodes with low-latency, high-bandwidth links among themselves. Hybrid applications, such as massively multiplayer games, may be concerned about both types of node attributes, e.g., low load for game logic processing and low latency to users for good interactive performance.

Given our target scenarios, we extract the following key requirements for any such resource discovery infrastructure. First, it must *scale* to large systems, consisting of thousands to tens of thousands machines while remaining highly available (the entire distributed infrastructure effectively becomes unusable by service operators if the resource discovery infrastructure becomes unavailable, much as the current Internet becomes unusable by service users if DNS becomes unavailable). Second, it must track both relatively static and frequently changing node characteristics. For instance, the system might track relatively static characteristics like operating system, available software licenses, and network coordinates [7, 18, 19], as well as more dynamic characteristics such as currently available CPU, memory, and disk resources. Third, the system must support an expressive query language that allows users to specify the ranges of resource quantities that their application needs, as well as how to select a utility-maximizing subset of available nodes when more match the requirements than are requested. Finally, the system should support queries over not just per-node characteristics such as load, but also over *inter-node characteristics* such as inter-node latency. Many scientific applications and Internet services require coordination or data transfer among nodes, so finding appropriate nodes for such applications requires specifying some minimum level of network connectivity, for instance placing bounds on maximum latency or minimum bandwidth between particular subsets of nodes (e.g., consider network services that must apply updates across a set of replicas, fine-grained parallel applications that must synchronize before proceedings with a computation, or a scientific application that requires high-speed access to a large data set). Supporting queries over such inter-node characteristics is particularly challenging because the information may be rapidly changing and scales with the square of the number of system participants. Thus the resource discovery service we envision combines aspects of distributed measurement, user utility specification, distributed query processing, and utility optimization.

A number of recent efforts have explored large-scale resource discovery [2, 4, 12, 14, 24, 28, 27, 8]. However, to the best of our knowledge, no existing system meets all of the above requirements. Thus, the principal contribution of this work is an exploration of the architectural space surrounding a resource discovery service with these target characteristics. To this end, we describe the design and implementation of SWORD, a Scalable Wide-Area Overlay-based Resource Discovery service. SWORD's main features are: i) a scalable, distributed query processor for satisfying the multi-attribute range queries that describe application resource requirements; ii) techniques for passively and actively balancing load in the range query infrastructure to account for skewed values in measurements; iii) an integrated mechanism for scalably measuring and querying *inter-node* attributes without requiring $O(n^2)$ time and space; iv) a heuristic optimizer that finds (approximately) utility-maximizing candidate sets of nodes matching user-specified per-node and inter-node requirements and utility functions; and v) mechanisms for limiting the running time and network resource consumption of the distributed query and optimization. We implement a variety of techniques—ranging from centralized to fully distributed to a hybrid approach—for distributing measurements, indexing them, and retrieving them. One interesting question we consider is the impact that these various distributed architectures have on end-to-end performance and network resource utilization.

Our evaluation shows that an architecture based on range searches in distributed hash tables can perform better than a "centralized" architecture for some workloads. Additionally, we find that a "hybrid" approach is promising, in which data is stored in a distributed hash table but the mapping of attribute ranges to the DHT node responsible for each range is stored in a central index.

Although we describe the resource discovery problem as a one-time only query-processing and optimization problem, the characteristics of the selected nodes and/or the node characteristics desired by the application may change over time. In these cases the application deployer would periodically reissue a SWORD query (automatically or manually), and the application would be migrated (automatically or manually) to the newly-selected set of nodes. Migrating a stateless or soft-state application (such as SWORD itself) is easy: the application is killed on the nodes that are no longer to be used, and is started on the newly-added nodes. Migrating a stateful application is more difficult, requiring either an application or OS-level process migration facility, or use of a virtual machine monitor that supports process migration.

SWORD is currently deployed as a continuously-running service on PlanetLab (http://www.swordrd.org). It tracks over 40 metrics per machine, collected from a combination of sources (ganglia [23], CoTop [20], and an implementation of the Vivaldi [7] network coordinates system), as well as inter-node latency.

The rest of this paper is organized as follows. Section 2 presents a high level level of the SWORD architecture and Section 3 describes the details of our implementation. We describe our evaluation infrastructure and our performance results in Section 4. We present related work in Section 5 before concluding in Section 6.

2. SYSTEM DATA MODEL AND ARCHITECTURE

The objective of SWORD is to allow end users (application deployers) to locate a subset of a global computation and communication infrastructure to host their application. Our approach is general to models where either the resource discovery infrastructure: i) is one of a number of systems monitoring global system characteristics and answering queries over that monitoring data, or ii) runs in isolation and also controls the allocation of resources to end users. Users begin by specifying requirements for a set of nodes. Resource specifications center around the notion of *groups* that capture equivalence classes of nodes with similar characteristics. For example, a content distribution service for streaming media might want several "virtual clusters" of nodes, with each cluster near one portion of its geographically distributed user base. Each cluster is an equivalence class. Each cluster would be composed of machines with enough disk space to collectively store all of the media files that users in the cluster's region might desire, and each cluster would have at least one link of sufficient bandwidth to one or more content creation sites so that new content could be quickly distributed to the clusters.

We assume that users will only want to deploy their application on a *subset* of the machines available to them–otherwise a node resource discovery system such as SWORD is unnecessary. A number of factors might motivate a user to limit the nodes on which they deploy their application. In a shared testbed such as PlanetLab, a user might simply want to be a "good citizen," limiting their application to the minimum set of nodes needed to accomplish the task. Alternatively, the platform might employ a computational economy in which users are charged for using node resources; in that case the motivation for a user to maximize her "bang for the buck" is obvious. Even in the absence of a financial or altruistic motivation, the user may find that her application actually runs slower when the deployment includes poorly-performing nodes than it does when the deployment uses a smaller number of nodes all of which are performing well; this "performance coupling" effect is common to applications with static work distributions or significant inter-node coordination requirements. Finally, the user may be evaluating the sensitivity of her application to different per-node resource constraints, network topologies, or network link characteristics. In that case she is interested not in maximizing the raw performance of her application, but in maximizing the closeness of match between the selected resources and an arbitrary experiment description; the goal is to find the closest embedding of her desired experiment configuration within the set of available nodes.

SWORD users specify a range of required and desired values of per-node and inter-node resource measurements, with varying levels of penalties (costs) for selecting nodes that are within the required range but outside the desired range. (In this paper we use the terms *cost* and *penalty* interchangeably to refer to the quantified "badness" of a choice.) For example, one application may desire 1 Mb/s of upstream bandwidth from all of its nodes, corresponding to a cost of zero. However, under constraint, the user may be satisfied with bandwidths greater than 512 Kb/s, with correspondingly higher cost. Bandwidths less than 512 Kb/s may be insufficient to support the application, corresponding to in-

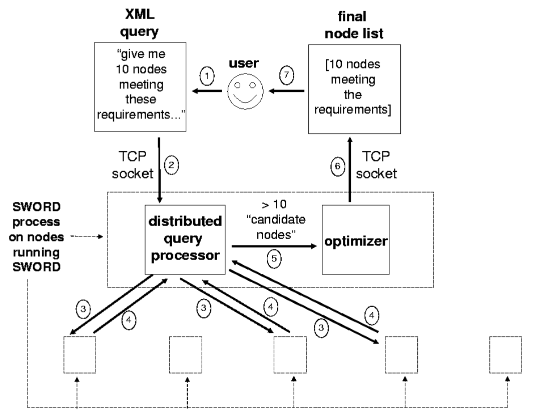

XML query
"give me 10 nodes meeting these requirements..."
1
user
7
final node list
[10 nodes meeting the requirements]
TCP socket
2
6
TCP socket
SWORD process on nodes running SWORD
distributed query processor
> 10 "candidate nodes"
5
optimizer
3
4
3
4
3
4

Figure 1: High-level architecture of SWORD

finite cost. SWORD endeavors to locate the lowest cost configuration that still meets the users requirements. While not explicitly explored as part of this work, a fundamental goal of SWORD is to also allow the specification of some (perhaps virtual) currency that the user is willing to pay for a particular configuration. A more sophisticated resource discovery infrastructure could use such values to perform adjudication among competing users during periods of high demand. SWORD itself, however, performs only resource discovery, not allocation. We expect it to run alongside a system that performs actual resource allocation.

Given a specification of user requirements, key architectural questions revolve around: i) tracking per-node and inter-node attributes, ii) determining the set of nodes to contact to resolve a particular resource discovery query, iii) optimizing the set of nodes returned to the user based on the request, and iv) maintaining scalability, availability, and load balance (of the resource discovery infrastructure) under a range of network conditions and query patterns. The rest of this section addresses these questions at a high-level, while Section 3 discuses them in detail.

An abstract representation of the SWORD system architecture appears in Figure 1. The user writes a query expressed in SWORD's XML query language syntax (step 1) and submits it to any node running SWORD (step 2). This query is passed to the distributed query processor component on that node, which issues an appropriate range query corresponding to the groups requested in the query (step 3). Although we have shown the query as being issued in parallel to SWORD nodes that may be storing data needed to answer the query, we have implemented and evaluated four range query mechanisms that use a variety of strategies for directing the query to the nodes that may hold relevant measurements. Once all of the results are returned from the distributed range query (step 4), the "candidate nodes" and their associated measurements are passed to the optimizer component on the node that originally received the user's query (step 5). The optimizer selects a utility-maximizing subset of the nodes returned from the distributed query and returns a list of them (along with the attribute measurements that led to their being selected) to the user (steps 6 and 7).

```
<request>
  <dist_query_budget>30</dist_query_budget>
  <optimizer_budget>70</optimizer_budget>
  <group>
    <name>Cluster_NA</name>
    <num_machines>4</num_machines>
    <cpu_load>0.0, 0.0, 1.0, 2.0, 0.01</cpu_load>
    <free_mem>256.0, 512.0, MAX, MAX, 1.0</free_mem>
    <free_disk>500.0, 10000.0, MAX, MAX, 5.0</free_disk>
    <latency>0.0, 0.0, 10.0, 20.0, 0.5</latency>
    <os>
        <value>Linux, 0.0</value>
    </os>
    <network_coordinate_center>
        <value>North_America, 0.0</value>
    </network_coordinate_center>
  </group>
  <group>
  <name>Cluster_Europe</name>
  <num_machines>4</num_machines>
  <cpu_load>0.0, 0.0, 1.0, 2.0, 0.01</cpu_load>
  <free_mem>256.0, 512.0, MAX, MAX, 1.0</free_mem>
  <free_disk>100.0, 10000.0, MAX, MAX, 5.0</free_disk>
  <latency>0.0, 0.0, 10.0, 20.0, 0.5</latency>
  <os>
      <value>Linux, 0.0</value>
  </os>
  <network_coordinate_center>
        <value>Europe, 0.0</value>
  </network_coordinate_center>
  </group>
  <constraint>
    <group_names>Cluster_NA Cluster_Europe</group_names>
        <latency>0.0,0.0,50.0,100.0, 0.5</latency>
  </constraint>
</request>
```

Table 1: Sample XML query.

2.1 Query Format

A SWORD query takes the form of an XML document with three sections. A sample query appears in Table 1, requesting two four-node clusters: one in North America and one in Europe. These clusters might be used by a computer animation studio to cache content that users download, with the expected user bases concentrated in North America and Europe. The machines in each cluster must have sufficiently low load and free memory to provide the service, and sufficient disk space to store the requisite files. We further assume that the nodes within a cluster perform cooperative caching of files, so that the service operator wants low network latency among all nodes in a cluster. Finally, we assume that the two clusters occasionally coordinate between themselves, and that therefore there must be at least one network link between the two clusters of sufficiently low latency for this coordination. In the remainder of this section we describe the SWORD query format and illustrate how its flexibility allows the animation studio example to be expressed as a query.

The first section of a SWORD query describes the (optional) resource consumption constraints the user places on evaluating the query. These constraints allow the user to trade reduced "quality" of the node selection for reduced network resource consumption in evaluating the distributed query and reduced running time of the optimization step in which candidate nodes are culled to a final approximately-optimal set. The **<dist_query_budget>** limits either the maximum number of SWORD nodes that may participate in answering a distributed query or the maximum number of candidates nodes that may be returned to the optimizer by the distributed query processor. The `<optimizer_budget>` limits the running time of the optimizer. In the example query, the user is allowing at most 70 nodes to be visited in processing the distributed query, and at most 30 seconds of running time for the optimizer.

The second section of the SWORD query specifies constraints on single-node and inter-node attributes of desired groups. In this paper we use the term "single-node attribute" for attributes that describe a single node in isolation—such as load, free disk space, number of network bytes sent during the last minute, or network coordinates—and the term "inter-node attribute" to describe attributes particular to pairs of nodes—such as latency, bandwidth, or loss rate. One set of single-node and inter-node constraints is associated with each of one or more *node groups*. All nodes within a node group have the same single-node and inter-node constraints, and the description of each node group also contains the number of nodes that should be in that group. In the example query, `cpu_load`, `free_mem`, `free_disk`, and `latency` are double attributes, `os` is a string attribute, and `network_coordinate_center` is a network coordinate attribute. By placing requirements on these attributes, the user has requested two groups: a cluster of four machines in North America and a cluster of four machines in Europe. The machines in each group must have load less than 2.0, at least 256 MB of free memory, and inter-node latency within each group of less than 20 ms. The North American nodes must be within a predefined network coordinate radius of a predefined "center" for North America and have at least 500 MB of free disk space, and the European nodes must be within a predefined network coordinate radius of a predefined "center" for Europe and have at least 100 MB of free disk space. We will explain the meaning of the second element of the `<value>` lines, and the meaning of the second, third, and fifth elements of the `<cpu_load>`, `<free_mem>`, `<free_disk>`, and `<latency>` lines, shortly.

The third section of the SWORD query specifies pairwise constraints between individual members of *different* groups. For example, our sample query specifies that there must exist at least one node in each group such that the latency between that node and at least one node in the other group is less than 100 ms. Although SWORD can handle any inter-node measurements, we use latency as the example in this paper. The assumption is that some external service measures these inter-node values and reports them to SWORD.

In SWORD, each requirement specifies a restricted form of a utility function. This family of utility functions is presumed to correspond to the Quality of Service (performance, reliability, predictability, etc.) that the application derives from different values of the various node attributes that affect the QoS. We describe these utility functions as "penalty functions," with "penalty" being merely the inverse of utility.

For double attributes, this penalty function has five regions: two regions of infinite penalty where attribute values are either too high or too low to be useful to the application, an "ideal" region of zero penalty, a constant-slope region of decreasing penalty towards the "ideal" region, and a constant-slope region of increasing penalty away from the "ideal" region. In resource discovery queries aimed at finding nodes for deploying an application, each attribute is likely to make use of only three regions, e.g., a user may specify that less than 128 MB of free memory is unacceptable, between 128 MB and 256 MB of free memory is acceptable but not ideal, and that more than 256 MB of memory is ideal. However, resource discovery queries can also be used to find nodes for evaluating an application under varying system conditions. For example, a developer may wish to determine how a service performs when half of its nodes become temporarily memory constrained. In that case, utility is maximized not by running the application on the set of nodes that maximizes the application's QoS, but rather from running the application on the set of node that most closely matches the experimental conditions that the developer is interesting in investigating. In this situation, the penalty curve's region of zero penalty might be between 128 MB and 256 MB, with increasing penalty in the regions 64MB-128MB and 256MB-320MB, and infinite penalty in the regions 0MB-64MB and more than 320MB.

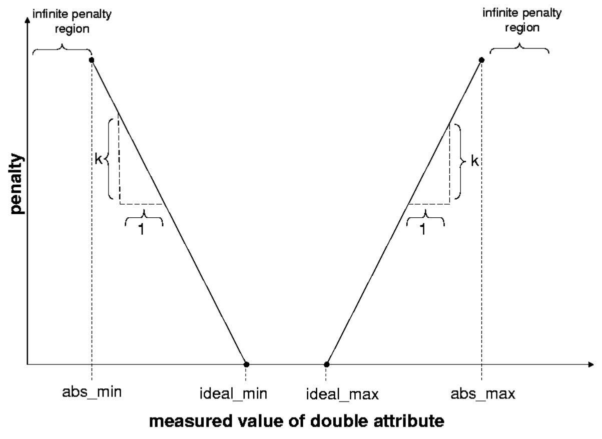


Figure 2: Penalty as a function of measured value of double attribute

Figure 2 illustrates the generic form of this curve, for the *double* requirement:
`<attr>abs_min,pref_min,pref_max,abs_max,k</attr>`
The purpose of the k is to give all the penalty graphs a consistent unit on the Y-axis. In other words, the k value serves the double-duty of expressing the relative importance the user places on the different attributes, and converting from the disparate units that different attributes have on the X-axis (bandwidth, CPU load, memory, etc.) into a uniform unit of "penalty." In the example query that appeared earlier, the user is indicating that each 0.01 deviation in CPU load from the ideal, each 1 MB deviation in free memory from the ideal, each 5 MB deviation in free disk space from the ideal, and each 0.5ms of inter-node latency deviation from the ideal, are all equivalent in terms of how much they hurt the application's utility.

Figure 3 illustrates the generic form of the penalty curve for the *string* requirement

```
<attr>
   <value>name1 name2 name3, p1</value>
   <value>name4 name5 name6, p2</value>
</attr>
```

This penalty curve associated penalty `p1` with values `name1`, `name2`, and `name3`, and penalty `p2` with values `name4`, `name5`, and `name6`. Any other values of the attribute are given an

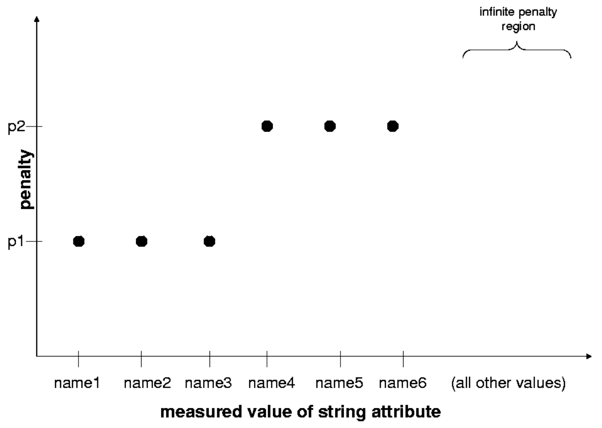


Figure 3: Penalty as a function of measured value of string attribute

infinite penalty.

Network coordinate attribute penalty curves are of the same form as string attribute penalty curves. Note that in addition to the network coordinate syntax that appears in Table 1, SWORD also allows queries that specify an explicit three-dimensional network coordinate center and radius. This is useful when the user knows the exact coordinates of the center of the region in which she is interested and the center does not correspond to a pre-configured location that SWORD recognizes, or when she desires a radius different from the pre-configured one for the area of interest.

The goal of the SWORD optimizer, then, is to minimize the sum of the penalties associated with the mapping of a subset of the nodes with non-infinite penalty, to the groups the user has specified, taking into account single-node penalties, inter-node intra-group penalties, and inter-group penalties.

2.2 Tracking and Querying Global Characteristics

It is relatively straightforward to build a single-node resource discovery tool that processes queries of the format just described. The tool takes as input a list of the available nodes and their associated single-node and inter-node attribute measurements, and a SWORD XML resource request. It then finds an appropriate set of nodes matching the description. While such a system is potentially useful, it is limited in its scalability and availability. In particular, all monitored nodes must periodically report their measured single-node and inter-node measurements to the machine running the resource discovery tool, and all users must contact that one machine for resource discovery requests. Assuming nontrivial metric update rates and query rates, a single node can become overwhelmed both in terms of CPU utilization and network bandwidth. Moreover, the failure of that node makes the resource discovery service unavailable. While these issues can be partially addressed by implementing the service on a cluster of nodes rather than on a single node, the cluster and the network link into the cluster are still points that can become overloaded or fail.

Thus, we set out to determine the conditions under which it makes sense to distribute the resource discovery process across multiple, cooperating, geographically diverse nodes. To this end, SWORD collects single-node and inter-node measurements from reporting nodes and stores them on a distributed set of server nodes. Although the reporting nodes and server nodes are logically separate, throughout this paper we assume they are the same set of nodes (i.e., SWORD runs on the same nodes that are being made available for users wishing to instantiate services and are therefore reporting measurements about themselves). To help distinguish the two roles a node can play, we use the term "reporting node" to indicate a node that is periodically sending measurement reports, and "DHT server node" to indicate a node that is part of the SWORD infrastructure and that therefore receives measurement reports and handles queries from users. We organize this latter set of servers using the Bamboo [21] structured peer-to-peer overlay network, although any other structured peer-to-peer overlay network, such as Chord [25] or Pastry [22], could also be used. Note that in this paper we refer to such systems interchangeably as structured peer-to-peer overlay networks and distributed hashtables (DHTs), but SWORD uses only their key-based routing functionality. On top of the key-based routing interface we build our own soft-state distributed data repository, the structure of which we will explain shortly.

For each of the n single-node attributes A_1, A_2, ..., A_n that can appear in a SWORD query, each reporting node periodically sends a tuple of *all* of its values for these attributes to n DHT keys k_1, k_2, ..., k_n, where each k_m is computed based on the corresponding value of A_m in a way we will describe shortly. Upon receiving such a tuple, a server (DHT node) stores the tuple in an in-memory hashtable indexed by the identity of the node that the report describes. For each attribute A, the range of possible values of that attribute is mapped to a contiguous region of the DHT keyspace using a function f_A. Thus, a list can be obtained of all nodes that are reporting A values in some range $x_{min} - x_{max}$ by visiting the DHT nodes that "own" all DHT keys between $f_A(x_{min})$ and $f_A(x_{max})$. For example, in the simplest case of a system where all attributes values are integers in the range 0 to the size of the DHT keyspace, f_A could simply be the identity function. This basic range search mechanism is alluded to in [15]. As we shall see in Section 3, the need to handle additional datatypes and limit the number of nodes that must be visited to satisfy a range query leads us to use more sophisticated mapping functions.

We assume that reporting and querying nodes are pre-configured with a default set of attributes and their corresponding f_A's. Additional attributes are added to the system when one or more reporting nodes pick a new attribute's name and f_A, and one or more querying nodes begin issuing queries using that f_A. One mechanism for distributing new attributes beyond those in the default pre-configured schema is SWORD itself; reporting nodes can include a list of the names of the non-default attributes they report, and the corresponding f_A's, as one of the pre-configured attributes that they send with each report. All nodes can periodically probe the full range of this attribute to retrieve all other node's schemas, or they can limit the query by node of interest or the time the node first added the attribute to its schema.

Using the range search primitive described earlier in this section, we can satisfy SWORD queries as follows. First, for

```
<cpu_load>0.0, 2.0</cpu_load>
<free_mem>256.0, MAX</free_mem>
<free_disk>100.0, MAX</free_disk>
<os>
    <value>Linux</value>
</os>
<network_coordinate_center>
    <value>North_America</value>
    <value>Europe</value>
</network_coordinate_center>
```

Table 2: Union group issued as range query for query in Table 1.

every group in the query, the querying node makes a list of all constrained single-node attributes and the region of each attribute's corresponding range where cost is non-infinite (utility is non-zero). Next, the per-group lists are joined into a single *union group* that contains, for each attribute mentioned in any of the per-group lists, a single range from the lowest to the highest non-infinite-cost values of that attribute among all groups. Table 2 shows the union group formed from the query in Table 1. Note in particular how the range search for the `<free_disk>` attribute is for values greater than 100.0, which is the widest of the two constraints in the groups (greater than 100.0 for `Cluster_Europe` and greater than 500.0 for `Cluster_NA`).

Finally one attribute is selected from the union group as the range search attribute. A distributed query is issued, that will visit all nodes that might contain reports about nodes whose value for the range search attribute is within the union group's range for that attribute. As each of those nodes is visited, only reports about nodes meeting all constraints in the union group will be returned. Recall that a node report contains all attributes for that node, making it feasible to do range searches based on any single attribute while returning only measurement reports that match all requirements in the union group. In our example query, we might select the `cpu_load` as our range search attribute; we then issue a distributed query that visits every node in the union group `cpu_load` range and returns from those nodes the identity (and reported attribute tuples) of every node whose reported value for `cpu_load`, `free_mem`, `free_disk`, `os`, and network coordinate meet the criteria in the union group.

The process described above indicates how single-node measurements are retrieved by the querying node. But the original query involves both single-node and inter-node measurements. Thus the next step is to obtain inter-node measurements. Because we expect inter-node measurements (latency, bandwidth, etc.) to change rarely compared to single-node measurements, it makes little sense to beacon them frequently to the soft-state DHT-based data repository. Instead, we leave those measurements on the nodes that take the measurements. Thus after retrieving all relevant single-node attributes from the distributed range search, the querying node contacts each node returned in that query to obtain the relevant inter-node measurements. (We will explain in Section 3 a mechanism SWORD uses, called "representatives," to avoid contacting all returned nodes for their inter-node measurements.) The single-node and inter-node results are joined and passed to the optimizer, which computes the utility-maximizing mapping of groups to nodes and returns that mapping to the user. Users may also choose to receive the single-node and/or inter-node measurements corresponding to the selected nodes.

3. IMPLEMENTATION

In this section we describe in detail how

- a SWORD node maps a measurement report (update) containing its single-node measurements to the server responsible for holding the update
- SWORD handles inter-node measurements
- a SWORD node issuing a query locates the set of servers responsible for range of the DHT keyspace corresponding to the attribute range to be searched
- the SWORD optimizer uses the user's XML query and the node measurements returned from the distributed query phase to find the (approximately) utility-maximizing set of nodes (and mapping of those nodes to groups in the user's query).
- SWORD balances load among server nodes in the face of a non-uniform distribution of reported node measurements

3.1 Reporting single-node attributes

The current SWORD implementation allows each attribute to be a double, a string, or a boolean. As mentioned earlier, each single-node attribute A is associated with a function $f_A(x)$ which maps a value from the range of A to a contiguous range of the DHT keyspace. This mapping happens in two steps. First, the value is converted to an integer within the key range of the DHT, between 0 and (2^{160})-1. Second, that value is mapped to an offset into one of N non-overlapping sub-regions of the DHT keyspace. For example, if N=4, the first two sub-regions are 0 to ((2^{160})/4)-1, and (2^{160})/4 to ((2^{160})/2)-1. These sub-regions serve to reduce by a factor of N the maximum number of servers that a query might need to visit. Each sub-region is responsible for zero or more attributes. Load balance among sub-regions is maximized when N is equal to the number of attributes M that the system tracks; in that case, one disjoint set of nodes is responsible for storing values for each attribute. If $M < N$ then some sub-regions will not store values for any attributes, while if $M > N$ then some sub-regions will store values for multiple attributes. No matter the relationship between M and N, the system will still perform correctly, though possibly at sub-optimal performance, and search performance will always improve as N is increased (ignoring any impact on update load balance).

Thus the f_A function provides two "passive" load balancing functions: it maps attributes to disjoint parts of the keyspace (and hence disjoint sets of server nodes), and it spreads potentially non-uniform value ranges to more uniform DHT key ranges. As an extreme example of the latter, consider a boolean value. A naive mapping would map "true" to one DHT key and "false" to another DHT key, leaving all other DHT keys corresponding to that attribute unused, and hence leaving some servers in that attribute's sub-region unused. A more clever mapping would use a single bit to represent the value and would randomize all other bits (except for the upper bits that map the attribute to a sub-region). Therefore one can think of the f_A function as

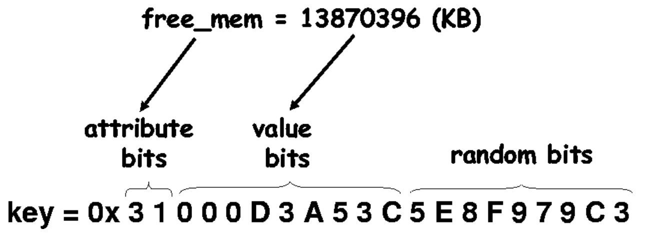


Figure 4: Mapping a measured value to a DHT key. This mapping is performed once per attribute in the measurement report, and the entire measurement report is sent to the calculated DHT key.

operating as shown in Figure 4, where the upper bits are determined by the identity of the attribute, the middle bits are determined by the value of the attribute being reported, and the lower bits are random.

The requirement that an f_A function be specified for every attribute that can be used as a range search attribute does not reduce the generality of the system, as default functions for the supported datatypes are supplied by the system. SWORD provides four default f_A functions:

- The default function for *booleans* maps "false" values to one range of the sub-region keys, and "true" values to a disjoint range of the sub-region. The lower bits are randomized so that not all "true" or "false" updates for a particular attribute end up on the same server.
- The default function for *strings* uses the ASCII representation of the string as the index into the sub-region; long strings are truncated, while short strings are appended with random bits. This scheme allows for prefix searches over strings, e.g., all strings beginning with "cat" can be located by searching the range from the sub-region offset corresponding to "cat" appended with all 0 bits to fill out the sub-region offset, to the DHT key corresponding to "cat" appended with all 1 bits to fill out the sub-region offset. Note that string range search is particularly convenient for searching IP address prefixes, assuming that the user has chosen to represent IP addresses as strings.
- SWORD offers two default functions for *doubles*: one for doubles that are percentages and one for doubles that are between 0 and MAX_DOUBLE. In the former case the range 0-100 is evenly spread to the size of a sub-region, and in the latter case the range 0-MAX_DOUBLE is compressed to $0 - Z$ where Z is the size of a sub-region.
- SWORD nodes may report their *network coordinates* [7, 18, 19]. We use a custom version of Vivaldi [7] for SWORD. The network coordinate in each dimension is treated as a standard double, and each node additionally publishes a synthetic "z-coordinate" attribute as a bit sequence for the explicit purpose of enabling range search over network coordinates. This attribute is computed as the z-coordinate of the n-dimensional (currently $n = 3$) network coordinates of the reporting node. This attribute is not returned during queries, but it enables multidimensional range search in the n-dimensional network coordinate space via a simple linear search over the z-coordinate attribute.

As mentioned earlier, users may add new attributes to the system at any time. They may choose to use one of the four default f_A functions, or they may define their own dynamically-loaded function. There is no requirement that all reporting nodes report the same set of attributes, only that all reporting nodes use the same f_A function for any given attribute A, and that querying nodes know about the f_A for attributes over which they wish to perform range search. The DHT servers do not need to know the details of the various f_A functions; the mapping of the update attribute to a DHT key is done by the reporting node, and the routing of the update to the server is done by the DHT routing infrastructure. When a report arrives at a server, the server records the update (containing all single-node attribute names measured by the reporting node and their measured values) in a hash table indexed by the reporting node. Our current implementation trusts users to supply well-behaved f_A functions when they add attributes to the system; SWORD does not currently protect itself against malicious user code that might infinite loop, cause an application crash, etc.

As we have described the system thus far, a node that reports R single-node attributes will periodically route the list of R measurements it has taken through the DHT to R distinct DHT keys, thereby enabling subsequent range search using any of those attributes. To reduce the number of messages sent per update, SWORD allows each attribute to be defined as a "key" or a "non-key." A report, containing the values of all key and non-key attributes for the reporting node, is sent to one DHT key *for each key attribute*. Thus only "key" attributes can be used as the attribute that directs the range search. For example, consider a system that defines the key attribute `load`, the key attribute `free_memory`, and the non-key attribute `MAC_address`. Users may search for nodes matching any conjunction of constraints over those three attributes, and they will receive a list of all nodes matching the constraints (and the values corresponding to all three attributes for each of those nodes), but SWORD will perform the range search using only the `load` or `free_memory` attribute. Although SWORD does not currently employ a query optimizer to choose which of the key attributes to use as the range search key, the assumption is that only a subset of report attributes will be useful in reducing the set of nodes to which a range query is directed. Thus, the example we gave is reasonable under the assumption that users will be primarily interested in constraining load or free memory in their queries.

Nodes can send their reports at a frequency interval of their choosing. In our current implementation, servers store reports in memory only, i.e., reports are soft state. The servers periodically time out these reports so that when a reporting node or its network link fails, that node will (eventually) no longer appear in the result set of any query. This soft-state-with-timeout mechanism requires that the re-publish interval must be no longer than the timeout interval; however, this seems reasonable, as node measurements are likely to change over fairly short timescales, making frequent re-publishing necessary for accurate query answering. In particular, using the reliable DHT storage layer rather than our memory-only storage adds unnecessary network, processor, and memory overhead by replicating each piece of data on multiple nodes, only to change that data soon thereafter. Note that our use of soft state also provides a low-overhead mechanism for recovery from failures within the range search infrastructure: if a DHT server fails, the next update that the DHT would have routed to that server will instead be routed to the new server now responsible for report. (All reports that had been stored on the DHT node that failed are lost.)

Because the data timeout interval may be several multiples of the update interval (to account for network message loss), and some measurement values may change substantially over short timescales, multiple versions of a report (containing measurements taken at different times) may simultaneously reside on different servers. If multiple reports from the same node are returned by the range search, timestamps on the reports are compared so that only the most recent report is returned. But note that the querying node may occasionally receive an out-of-date report that it cannot determine is out-of-date. For example, reporting node R may send update R1 with attribute a=1 and later send update R2 with attribute a=5. If querying node Q queries for nodes with a=[0,3], it will receive report R1 because the R2 report with the more recent timestamp is not within the requested range query range. However, the maximum staleness of a result is bounded at all times by the report timeout interval. For example, in our PlanetLab implementation, the update interval is 2 minutes and the timeout interval is 5 minutes, so no query will be answered with data that is more than 5 minutes out of date.

3.2 Distributed range search for single-node attributes

As discussed in Section 2, an XML resource discovery query with N query attributes is converted to a query over the conjunction of the N ranges of attributes in those queries. For each attribute, the range that goes into this "union group" is the widest range necessary to find all possible nodes that could satisfy the constraint for any group. One of the attributes in the query, which must be a "key" attribute, is selected as the range search attribute. In our current implementation, this attribute is selected randomly, and can be overridden explicitly in the query. However, there is a clear opportunity to optimize the selection of the range search attribute; the goal of such an optimization would be to pick the attribute from the query that will require the query to visit the fewest number of servers (perhaps attempting to simultaneously avoid overloading any particular sub-region across queries).

Once the search attribute A is selected, f_A is applied to the lower and upper bounds of the constraint range to find the lowest and highest DHT keys corresponding to the range of the constraint: $key_{low} = f_A(constraint_{min})$, $key_{high} = f_A(constraint_{max})$. As a special case, if network coordinates are selected as the range search attribute, then the range of the DHT keyspace that is searched is the contiguous range between the smallest and largest z-coordinate to which the multi-dimensional coordinate ranges in the query map.

The full query is sent to the nodes in the computed DHT key range. Each node receiving the query applies all constraints in the query to the nodes whose reports it holds, and sends back to the querying node all node reports that match the conjunction of constraints in the query. We have implemented and evaluated three range search techniques. These three mechanisms offer a variety of tradeoffs in performance, robustness, and the network resources consumed in satisfying the query.

- *Leaf-set walk:* Figure 5 summarizes this approach. The query is first routed to the server responsible for a randomly generated key between key_{low} and key_{high}. That node then clones the query and forwards one clone (via Bamboo's UdpCC) to each (four, by default) successor node in its leaf set. Each subsequent node forwards the query to the farthest successor node in its leaf set. This process continues until the query reaches a node that is outside the query range. When the query reaches the node that owns the maximum DHT key in the range, that node forwards the query to the node that owns the minimum key in the query range. The query completes when it returns to the node that initially received it. Thus the query eventually visits all nodes in the range, with four nodes being visited in parallel.

  A query can limit the number of servers visited in two ways. First, the query can directly specify the maximum number N of servers to be visited. In our implementation, the query is dropped when the query has visited $N/4$ servers (since each clone of the query visits one fourth of the nodes in the desired range). Second, the query can specify the maximum number M of candidate nodes it desires to visit. In this case, each server asks the querying node whether it has received enough candidate nodes yet, and only forwards the query to the successor if the querying node has received an insufficient number so far. This mechanism can be thought of as a callback that the querying node requires at each server hop before allowing the forwarding of the query to the next DHT server node.

- *Routing table walk:* This approach is similar to the "leaf set walk," but instead of exploring the transitive closure of the successor set pointers and pruning paths that fall outside the range, the "routing table walk" explores the transitive closure of the routing table pointers and prunes paths that reach outside the range. This approach is best illustrated by example. First, suppose the query range is 1230000 - 1238fff. The query is routed through the DHT to a random node in the range of the common prefix, which in this case is a node whose DHT key starts with 123. Then, that node sends a message to all relevant one-digit extensions, e.g., 1230, 1231, 1232, 1233, ..., 1238. The message instructs those nodes to contact all nodes with that prefix [11]. By extension, the range 1231000-1232500 can be reached as follows. The query is first routed to a node whose DHT key starts with 123. That node sends two message: a "contact all with prefix" message as in the previous example is sent to a node with prefix 1231, and a "contact all nodes less than 12325" message is sent to a node with prefix 1232. The latter node sends a "contact all with prefix" message to nodes with prefixes 12320, 12321, 12322, 12323, and 12324, and a "contact all nodes less than 123250" message to a node with prefix 12325. In practice, only a single message type is used, specifying a prefix representing the set that needs to be reached, and the upper and lower bounds of the query.

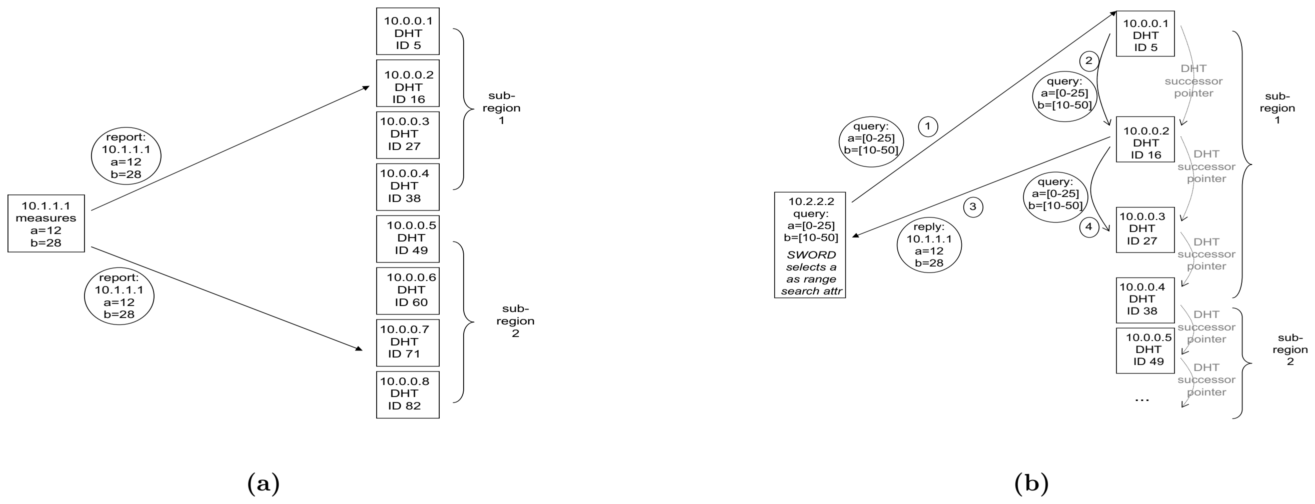


Figure 5: Reports and queries using the leaf set range search strategy. In (a), a copy of each measurement report is routed to one DHT key for each "key" attribute in the update. In (b), a querying node selects a random node in the range for attribute a. In this example, the second of three visited DHT nodes contains values matching the query and returns those values in parallel. Note that steps 3 and 4 take place in parallel and that the third node has no matching values in this example. We have omitted a final step in which node 10.0.0.3 notices that it owns the top of the query's range and sends the query back to 10.0.0.1, from which the "search complete" message is sent to the querying node. Also, we have drawn the strategy as if only one successor is followed out of each node, but SWORD actually follows all four of each Bamboo node's successor links in parallel to reduce query latency.

The main benefit of this approach is that, depending on the query and the routing base of the DHT, it can exploit more parallelism than can the leaf set approach. Additionally, this approach would work with a DHT that does not use leaf sets, such as Tapestry. The main drawback of this approach is that the amount of parallelism is unconstrained; therefore it requires a callback mechanism like the one we implemented to limit the number of candidate nodes returned in the "leaf set walk" if it is desired to avoid flooding the network quickly if the search range is broad.

- *Hybrid:* The third range query approach, summarized in Figure 6, decouples the process of identifying the set of servers to visit from the process of visiting those servers. In this "hybrid" approach, every node in the DHT periodically reports to a central server the range of the keyspace for which it is responsible. Each query is routed first to the central server, which can immediately identify the full set of DHT nodes that the query must visit. The central server forwards a copy of the query to those nodes simultaneously. Thus the query is always satisfied in three hops, as opposed to the leaf-set walk approaches, in which the number of hops scales with the width of the query range and the number of nodes in the system. Note that the "central server" we refer to here may actually be composed of multiple machines, perhaps even geographically distributed. We refer to it as a "central server" to indicate that it runs on well-connected, dedicated infrastructure machines that do not participate in the DHT.

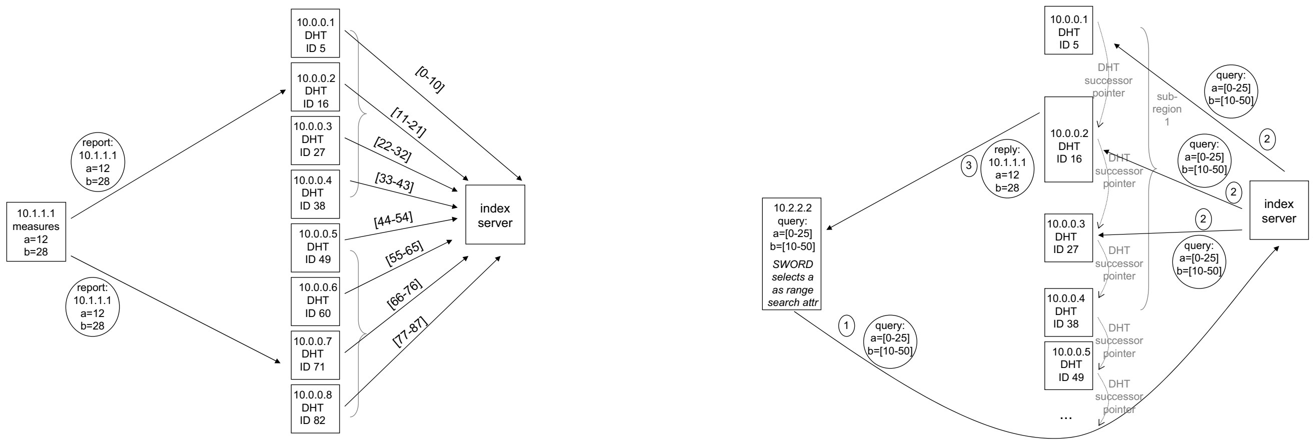
In comparing the "hybrid" approach to the pure-DHT approaches, one can make an analogy to Napster and Gnutella. Napster, like the hybrid approach, stores the index on a central server but stores data (node measurement reports) on the distributed nodes. Gnutella, like the DHT approaches, combines querying and routing, and does not build an external index. The cost, performance, scalability, and reliability tradeoffs between our hybrid and DHT approaches are expected to be similar to those between Napster and Gnutella.

After using one of the three range search mechanisms described above, the SWORD distributed query processor contacts via TCP the representatives indicated by the set of returned nodes to retrieve the pairwise inter-node measurements between nodes in the returned set.

3.3 Inter-node attributes

The aforementioned process retrieves the identities, and all single-node attributes, of reporting nodes that meet the *single-node* constraints in the original query. Unlike existing resource discovery systems, SWORD adopts the philosophy that users may wish to take *inter-node* attributes, such as latency, bandwidth, and loss rate, into account when selecting nodes. Recall that the optimizer (and query semantics) requires knowledge of both the single-node and inter-node attributes of candidate nodes. SWORD could handle inter-node attributes similarly to single-node attributes by having all reporting nodes measure and send all of their inter-node measurements along with their single-node measurements, and retrieving those measurements in the distributed range query. Instead, SWORD's handling of inter-node attributes differs from its handling of single-node attributes in two ways.

First, to reduce bandwidth usage, measuring nodes store their inter-node attributes locally rather than routing them to a remote server. Because we are aware of no efficient



(a) **(b)**

Figure 6: Reports and queries using the hybrid range search strategy. In (a), each DHT node periodically informs the index server of the key range for which the DHT node is responsible. In (b), a query is sent to a logically centralized index infrastructure (drawn here as a single machine) that tracks the range of DHT keys each node in the DHT is responsible for. The index infrastructure receives each user query and forwards a copy of it to each DHT node in the appropriate range in parallel. Nodes with matching values for all attributes in the query (a and b in this example) reply directly back to the querying node. Measurement report updates are handled as in Figure 5

mechanism to perform range search over pairwise measurements, there is no apparent benefit to storing those measurements in the DHT. We expect that the range search over the single-node attributes will sufficiently reduce the search space so that contacting all nodes that qualified based on their single-node attributes can be done in reasonable time. (And indeed our experimental evaluation shows this to be true.)

Second, to reduce measurement overhead, only a subset of nodes, called "representatives," measure (and store locally) inter-node attributes; they act as representatives for "nearby" nodes that are assumed to possess similar inter-node measurements. In a system with N nodes, the approach we have described thus far requires N^2 measurements and N^2 storage space for inter-node attributes. To reduce resource consumption while still allowing high-precision distributed queries over inter-node attributes, SWORD leverages the observation that nodes typically fall into a number of equivalence classes for inter-node attributes. For example, the bandwidth between node A in Autonomous System 1 and node B in Anonymous System 2, is likely to be similar to the bandwidth between any node in Autonomous System 1 and any node in Autonomous System 2. SWORD therefore allows arbitrary groups of nodes to define a "representative" responsible for measuring and storing the inter-node attributes between that group of nodes and all other such groups. The mapping from each node to its representative is one of the single-node attributes it reports periodically; this is essentially an "object location" pointer indicating where to find the holder of that node's inter-node measurements. After a querying node receives the node reports in response to its range search, it will query (via TCP) all nodes that serve as representatives for nodes in the returned set, asking for the inter-node attributes between all pairs of nodes in that returned set. We anticipate that inter-node measurements will be taken less often than single-node measurements because they are likely to change less frequently, and because the overhead of measuring between all pairs of representatives can make frequent measurement prohibitively costly. Thus, representatives store inter-node data persistently to protect against data loss in the event that the representative fails. SWORD currently makes no attempt to replicate inter-node measurements across nodes or to enable failover from one representative to another, though this is clearly possible through traditional leader election and data replication techniques.

Choosing appropriate representatives is an orthogonal area of research that might leverage existing work on network-aware clustering [16]. While we intend to pursue this model as future work, for the purposes of this paper, we assume some appropriate scheme for choosing representatives that will reduce the number of nodes whose inter-node characteristics must be stored by some constant factor. For bootstrapping, each node in the system need only know the identity of its representative. In addition to periodically reporting as a single-node attribute the identity of its representative, each node also reports whether it is itself a representative. When a node starts up, if it is a representative, it performs a distributed query on the "is a representative" attribute. This query retrieves the identities of all representatives in the system. The querying node subsequently takes periodic measurements of the desired inter-node attributes to all all other representatives in the system, and stores them locally.

3.4 Optimizer

Independent of the technique used for performing the search on single-node attributes, the result is a list of all nodes that satisfy the union of all single-node constraints in the user's original query. The querying node then combines this information with the appropriate inter-node measurements from the representatives, as described in Section 2. The next step is to use this information to determine an appropriate mapping of nodes to the groups specified in the original query. Creating the groups of a specific size such that all links have a latency less than the stated maximum is an NP Hard problem; it is an instance of the *k-clique* problem.

A naive algorithm for solving this problem is to enumerate all possible combinations of disjoint groups that meet the latency requirements. The running time of this algorithm is exponential. To add further to the complexity of this problem, we have the extra constraint that the resulting groups must meet cross-group latency requirements. Further, we want to find the set of groups that *best* satisfy the requirements, as opposed to just *any* set of groups that satisfy the requirements. Enumerating all possible solutions and finding the optimal combination is potentially a very time-consuming, computationally intensive task.

Therefore rather than check every possible combination, we use pre-sorting and pruning techniques to find a good group combination without searching every possible combination of groups. We first compute the "single-node costs" for each candidate node, i.e., for each candidate node and group, the cost that would be incurred if that node were to be placed in that group. Once the single-node costs are computed for each node in the candidate list, the group finding process begins. Starting with the first node in the candidate list and the first group, we remove any other node in the list that does not have a pairwise latency to the first node within the acceptable range. We add the first node to our group list, and continue this process with the second node in the candidate list until an acceptable group is found or the candidate list size is too small to ever satisfy the request. This is analogous to doing a depth first search in a binary search tree. If we detect that the candidate list is too small to continue, we prune the remainder of the search path, and begin removing nodes from the group list while moving back up the tree until we have returned to a point where the list is not too small. We then continue searching an alternate branch down the tree in the same manner. This tree searching technique is a hybrid between a depth first search and a breadth first search that detects bad paths early by keeping track of the candidate set size. This is a more efficient algorithm than a standard depth or breadth first search that must reach a leaf node before trying an alternate path starting from the root.

Pre-sorting the candidate node list based on single-node costs establishes a cost-based order to the candidate node list for each group. This allows the nodes that best meet the requirements for a particular group to be evaluated first. If the user-specified optimizer budget is low (which means the user does not want to spend a long time waiting for the optimizer to complete), this pre-sorting increases the chances of finding a low-cost node assignment for that group group early in the search. Further, for each group in the query, we maintain an ordered list of potential membership sets for that group, based on the potential membership set's total cost (determined by summing the single-node costs for each group member). The membership set with the lowest total cost is the best mapping of nodes for the group in question.

The initial phase of the optimizer described above uses a search tree to find a cost-sorted list of every possible group that meets the requirements specified for each requested group. The final step of the optimizer is to evaluate these lists to check for cross-group constraints and find the optimal group combination out of all possibilities. In many cases the optimal combination is the first combination, since the candidate group lists were already pre-sorted based on aggregate cost. This search continues until all possibilities are evaluated, or for the remainder of the user specified optimizer budgeted search time.

Pseudocode for our algorithm is shown in Table 3.

3.5 Load Balancing

A potential drawback of the three distributed range search techniques we have described is that both updates and queries are likely to be non-uniformly distributed. Yet DHT node identifiers are created uniformly at random, so we will be attempting to map a nonuniform distribution of reports and queries to a set of servers whose keyspace responsibilities are uniformly distributed. This will lead to load imbalance.

As a simple example, if reporting nodes are concentrated in one geographic region, they may be more heavily used during daytime hours in that geographic region, and their various utilization statistics will therefore tend to be high at those times. Furthermore, most queries are likely to request nodes with low utilization attributes. As a result, both updates and queries may have a tendency to be skewed towards a small subset of SWORD nodes.

We address the load balancing of updates through a combination of passive and active techniques. We discussed in Section 3.1 the passive techniques, namely the use of subregions, f_A functions that are customized to the range of values of each attribute A, and randomized lower bits of the key to spread discrete-valued measurements to a continuous range.

For active load balancing, we use the technique described by Karger and Ruhl [15], which operates approximately as follows. Each SWORD node periodically generates a random key and sends its load to that DHT key. If there is sufficient load imbalance between the sender and receiver, the less heavily loaded node moves to take over some of the load (i.e., some of the keyspace) from the more heavily loaded node. Note that we do not explicitly transfer any keys when a node moves. Instead, our soft-state mechanism combined with the DHT's self-healing property ensures that the stale data will be aged out and the new mapping will be used for subsequent updates. In our implementation, a node "moves" in the DHT keyspace by terminating itself (the Bamboo process) and restarting the Bamboo process using the new DHT key.

We currently use update load as our only load metric for active load balancing; we make no attempt to balance query load. While it is perhaps also important to account for query load, it is more difficult to balance query load using the Karger mechanism. Consider the case where most queries are looking for loads between 0 and 1, and these correspond to DHT keys (after expansion) between 1000 and 3000. Suppose there is currently a node with ID 2000 that is responsible for that entire range (Bamboo maps keys to the node with the closest ID). Suppose we move a node with some key outside this range to ID 2500. Users are still searching for keys between 1000 and 3000 to satisfy their queries, so the load imbalance can actually become worse—now two nodes

```
public grouplist findGroups {
  foreach group {
      while(!stop) {
          //1:  Find group of specified size
          newGroup=Checklatency(node_list, 0, newVec,
                    size, min_latency, max_latency);
          if(newGroup.size()=size)
              //2a:  Group was found
              newGroup.computeCost();
            groupList.add(newGroup);
            groupList.sortByCost();
          else
              //2b:  Reached end of search
              stop=true;
      }
  }

  if(groupList.size()!=0)
      //3:  Check for constraints between groups
      done=CheckConstraints(groupList);
      if(!done)
          //4:  No link found. The request cannot be satisfied
          return null;
      else
          return grouplist;
  }
}

//Recursive algorithm for checking latency
public nodeList CheckLatency(listNodes, count,
        newVec, max, min_latency, max_latency) {
  oldList=listNodes;   //Make copy of list
  source=oldList.pop(); //Get first element

  //Find links with desired latency
  for (i=0 to oldList.size()) {
      dest=oldList.elementAt(i);
      latency=latencyMap(source, dest);
      if (latency < max_latency
        && latency > min_latency) {
            newList.add(dest);
      }
  }

  if(newList.size() < max-count-1) {
      //Not enough nodes to continue on this path
      if(oldList.size() < max-count) {
        if(stack.size() > 0 && count > 0) {
            newList.removeAllElements();
            count--;
            newVec.removeLastElement();
            oldList=stack.pop();
            return CheckLatency(oldList, count, newVec,
                max, min_latency, max_latency);
        }
        else
            return null;
      }
      else
          //Continue down same path
          newList.removeAllElements();
          return CheckLatency(oldList, count,
              newVec, max, min_latency, max_latency);
  }
  else {
    newVec.add(source);
    stack.push(source);

    if(count==max)
      return newVec;
    else
      return CheckLatency(newList, count, newVec,
          max, min_latency, max_latency);
  }
}
```

Table 3: Pseudo code for our optimizing group finding algorithm.

are overloaded (node 2000 and node 2500) instead of just one.

We note that the Karger and Ruhl load balancing technique is not without its drawbacks. Perhaps significant is that the logarithmic-hops routing property of DHTs depends on nodes being assigned keys drawn uniformly at random from the DHT keyspace. The Karger approach changes this distribution, making it denser than uniform random in popular regions of the keyspace and sparser than uniform random in unpopular regions. This will affect the average path length between some hosts in the DHT, and it will increase routing load in "sparse" parts of the address space. Our results indicate that the inflation in path length is acceptable for the networks that we consider. For networks of sufficient size where this additional overhead becomes significant, related work [4] shows how to augment a DHT to account for such skewed assignment of nodes to the DHT keyspace.

3.6 PlanetLab

At the time of this writing, SWORD has been running continuously on over 200 PlanetLab for multiple weeks. (Constraints of the Bamboo DHT prevent us from running Bamboo, and therefore SWORD, on nodes that are connected only to Internet-2, but we are running SWORD on all other accessible PlanetLab nodes). The architecture of SWORD on PlanetLab is the same as that illustrated in Figure 1. A SWORD instance runs on every PlanetLab node. To issue a query, a user makes a TCP connection to any SWORD instance, and sends the XML query. The contacted SWORD instance initiates the distributed range search, followed by the retrieval of the needed inter-node measurements and invocation of the optimizer. That node then returns to the user over the same TCP connection the result (a list of nodes, the groups to which they have been assigned by the optimizer, and the raw node measurements that were used in making the assignment). For PlanetLab, we extended our query language to allow users to specify a per-group maximum number of nodes that can be assigned to that group from any single PlanetLab site.

We designed the mechanism for updates with extensibility in mind. In particular, we wanted to make it easy to add new attributes and data sources, without having to restart any nodes. Updates are handled as follows. Every two minutes, each node reads a configuration file that states for each attribute its name, its type, the name of the Java class containing its f_A function, and the name of the data file that will contain its current value. A separate process, run from cron, ensures that the data file for each attribute is kept up-to-date (at any frequency interval, which may be longer than two minutes). It then reads the data sources themselves and sends out a measurement report.

To make this architecture more concrete, we currently utilize two data sources (in addition to the network coordinates data source that is built into SWORD): the ganglia [23] daemon running on each node, and the "CoTop" tool [20] running on each node. Every two minutes a cron job on each node invokes a script that contacts the local ganglia daemon running on port 8649 of that node, parses the returned metrics relevant to that node (in some Ganglia configurations, metrics for all nodes at that PlanetLab site will be returned, rather than only metrics from the local node), and makes them available to SWORD. Likewise the

CoTop tool is invoked every two minutes on each node to collect information about the resources being used by the most resource-consuming process on that node. Likewise every two minutes SWORD reads the current ganglia and CoMon metric files, creates an update message containing all metrics, and sends the update message to the DHT nodes corresponding to the values of the "key" attributes in the update.

To add a new metric, the administrator simply writes a script that both measures the new metric of interest and writes it to a file. Then she adds a line to the cron configuration file on her machine to invoke the script at the interval at which she wants the metric to be measured. Finally she adds a new line to the SWORD configuration file, specifying its name, its type, the name of the Java class containing its f_A function, and the name of the data file that will contain its current value. At the next two-minute interval, SWORD will re-read the configuration file and incorporate the new metric into the update it generates; the SWORD service does not need to be stopped and restarted for the new metric to be added.

Our PlanetLab implementation does not use "representatives" or direct measurement of inter-node latencies. Instead, inter-node latencies are estimated using the network coordinates of the pair of nodes between which the latency measurement is desired.

The attributes that are currently available to be queried from SWORD appear in Table 4.

4. EVALUATION

4.1 Overview

In our evaluation we were interested in answering the following questions:

1. How does performance scale with number of nodes, when workload remains constant?
2. How do the query rate and report rate affect performance? How do the different range search approaches we have implemented (leaf-set walk, routing-table walk, and hybrid) compare to one another and to a centralized implementation?
3. Can our DHT-based approaches heal quickly from failures, even large ones?
4. How much network bandwidth does a report and query workload consume?
5. How much of an impact does the skew of node DHT keys resulting from load balancing have on the length of DHT routing paths for updates?
6. What is the end-to-end performance of queries, including time to retrieve inter-node statistics from representatives and to pass candidate nodes through the optimizer?

We evaluated SWORD on two comparable clusters: one cluster of 38 IBM xSeries PCs with Dual 1 GHz Pentium III processors and 1.5 GB of RAM, and one cluster of 38 Dell PowerEdge 1750s with 1GB RAM and single 2.8GHz Intel Xeon processors. Both clusters used Gigabit Ethernet. We used ModelNet [26] to emulate wide-area bandwidth and latencies, and the Inet topology generator [5] to

Table 4: Attributes currently measured and queryable by the SWORD PlanetLab deployment.

attribute name	type	units
hostname	string	
cpu_aidle	double	%
ip	string	
mtu	double	B
cpu_nice	double	%
cpu_speed	double	MHz
cpu_user	double	%
mem_shared	double	KB
load_fifteen	double	
cpu_idle	double	%
proc_total	double	
mem_cached	double	KB
proc_run	double	
cpu_num	double	
pkts_in	double	packets/sec
part_max_used	double	%
swap_total	double	KB
bytes_out	double	bytes/sec
load_five	double	
os_release	string	
machine_type	string	
gexec	string	
disk_total	double	GB
boottime	double	s
mem_total	double	KB
disk_free	double	GB
mem_buffers	double	KB
cpu_system	double	%
bytes_in	double	bytes/sec
os_name	string	
sys_clock	double	s
swap_free	double	KB
load_one	double	
pkts_out	double	packets/sec
mem_free	double	KB
latency	double	ms
gnp	network_coordinates	n/a
cotop_txhog	string	(slice name)
cotop_prochog	string	(slice name)
cotop_txkb	double	Kb/s
cotop_nprocs	double	
cotop_rxhog	string	(slice name)
cotop_memhog	string	(slice name)
cotop_rxkb	double	Kb/s
cotop_mempercent	double	%
cotop_cpupercent	double	%
cotop_cpuhog	string	(slice name)

create a 10,000-node wide-area AS-level network with a variable number of client nodes (225, 450, and 900) always with 4 client nodes per stub. Transit-transit and transit-stub links were 45 Mb/sec and client-stub links were 1.5 Mb/s. Latencies were based on the Inet topology. We chose one client node at random from each stub to serve as the "representative" for itself and the other nodes in its stub. We used the Bamboo DHT version available in December, 2003.

Our baseline workload consists of updates (measurement reports) and queries, each issued by each of 900 virtual nodes. Each node issues one update every five minutes, with ten attributes per update: eight doubles (three for the network coordinate space) and two strings. Each node issues one query every five minutes. In order to stress the system, each query uses the same range search attribute (thereby restricting queries to one sub-region). We use eight subregions (as defined earlier) overall. We used a Zipf workload, defined as follows.

For *reports*, Doubles are Zipf distributed between 0 and 1000, with the heavy portion of the distribution at 0. Network coordinates are reported as measured by a network coordinates subsystem running in the same address space as SWORD and taking measurements of the emulated network topology; those three double values tend to be in the range -100 to 100. One of two strings is chosen for each string attribute, and the choice is made using a Zipf distribution.

For *queries*, each query specifies the conjunction of three constraints, one for each of three of the double attributes, with each of the three constraints identical within (but not between) each query. The width of the query ranges is chosen uniformly at random, centered at a value that is Zipf-distributed, with the heavy portion of the distribution at 0. Note that our Zipf report/query distribution is designed to stress the system, as the aggregate update and query load will be spread among a smaller number of servers than it would be with a uniform update and query load distribution.

The general justification for this workload is as follows. Uniform random query width models the "picky-ness" of different users being uniformly randomly distributed. Zipf distribution for the low point of the query models a user bias toward picking the "best" machines (which we assumed was represented by the low end of the range). We biased Zipf for the updates toward the same end as Zipf for the queries because i) we wanted to ensure that enough candidate nodes were actually returned (the evaluation would be meaningless if the user queries matched only a trivial number of nodes), and ii) one would assume that users would not want to use a system where their desires could rarely or never be satisfied. Had we chosen values in updates to rarely be satisfiable, our system would perform better than we report here. We use a Zipf parameter of 0.9.

We examined four range search configurations. (i) The **leaf-set walk** approach was described in Section 3. By default we do not limit the number of hops or number of nodes returned. (ii) The **routing-table walk** approach, also described in Section 3. We measured this approach both with callbacks to the querying node before taking a hop to the next server, and without callbacks (i.e., a flood of the query subject to the algorithm described in Section 3), but for the workloads we used, the two approaches showed similar performance, so we present only the non-callback results here. (iii) A logically **centralized** configuration. In this scenario

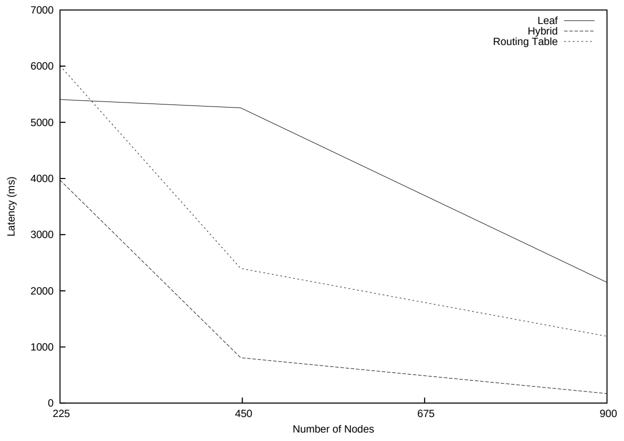


Figure 7: Median range search latency versus number of nodes for our three non-centralized range search techniques.

there are 8 or 64 servers to which all updates and queries are directed. The DHT keyspace range is statically partitioned evenly among the servers, analogous to range partitioning in parallel databases and the way in which the keyspace is automatically partitioned in a DHT. Updates and queries are then handled analogously to the way they are handled for the leaf-set walk and routing-table walk approaches, except that updates go to one of the (8 or 64) servers and queries are sent in parallel to the servers that are responsible for the portions of the range indicated in the query. For evaluating this approach, we modified the emulated topology so that $N/4$ groups of four servers each were connected via a 45 Mbps, 1 ms-latency network link to their upstream transit node, where N is the number of "central" servers (8 or 64). This was done to emulate an environment in which a service provider has placed the N servers in N/4 geographically distributed, well-connected collocation centers. (iv) A **hybrid** configuration, as described in Section 3), with 8 index servers. The index servers' network links are configured as described above for *central*. We found that the performance of hybrid was essentially identical regardless of the number of nodes serving as index servers (up to 64, the maximum that we tried), so we report only the results from the 8-index-server runs here.

4.2 Results

4.2.1 Range search scalability under constant workload

Figure 7 shows how the performance of our three non-centralized range search techniques scales with increasing number of nodes, given a fixed workload. The main effect we expect to see here is improving performance as a larger number of nodes are available to "sink" the update load, leading to less contention for node resources (including last-hop bandwidth) when queries are issued. This is the case in the hybrid and routing-table walk cases. The effect is less pronounced in the leaf-set approach because the reduction in per-node update load is largely offset by an increased number of nodes that must be visited to satisfy the range query (because a larger number of nodes in the DHT means

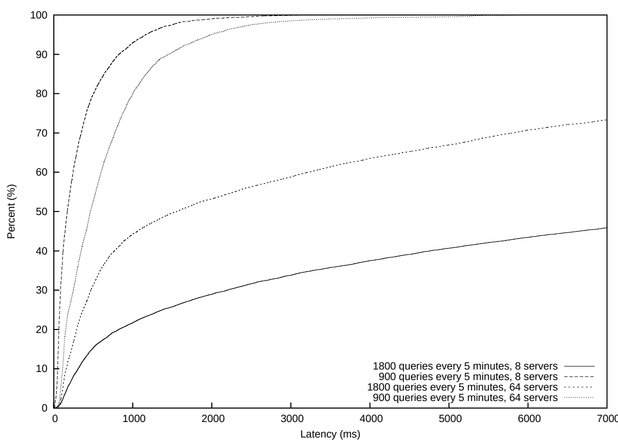


Figure 8: Cumulative distribution function of query latency, for workloads of 1800 queries every 5 minutes and 900 queries every 5 minutes, using the centralized approach with 8 and 64 servers.

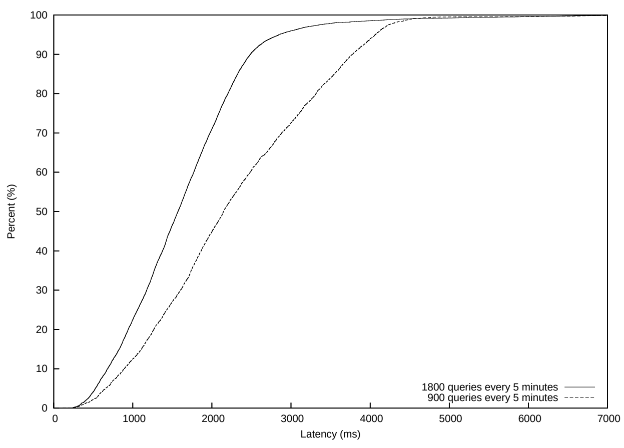


Figure 10: Cumulative distribution function of query latency, for workloads of 1800 queries every 5 minutes and 900 queries every 5 minutes, using the leaf-set walk approach.

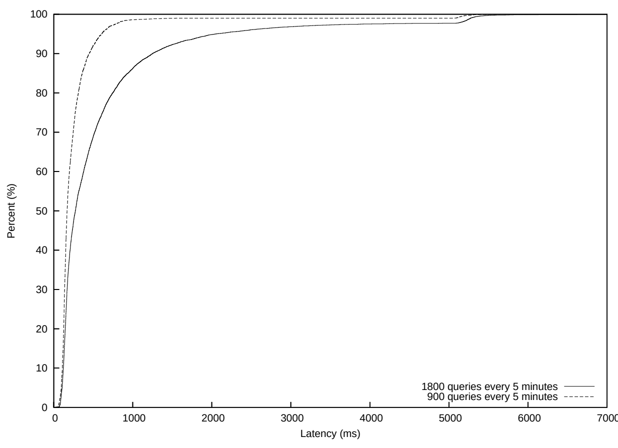


Figure 9: Cumulative distribution function of query latency, for workloads of 1800 queries every 5 minutes and 900 queries every 5 minutes, using the hybrid approach.

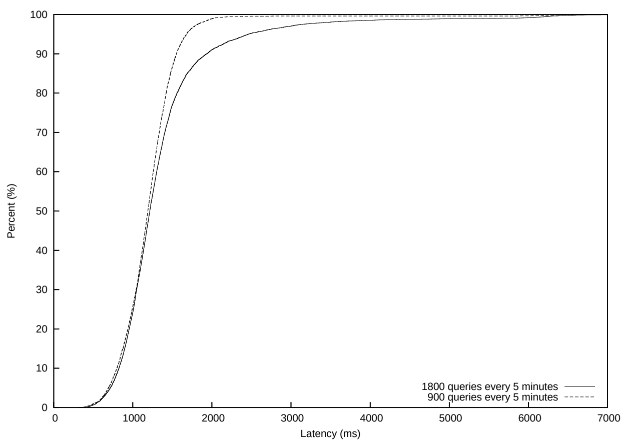


Figure 11: Cumulative distribution function of query latency, for workloads of 1800 queries every 5 minutes and 900 queries every 5 minutes, using the routing-table walk approach.

each node is responsible for a smaller range of values). Although more nodes must also be visited in the routing-table walk search strategy as the number of nodes increases, the routing-table walk approach is able to exploit a greater degree of parallelism (a routing table out-degree of 16 versus a successor set out-degree of 4 in Bamboo) and so is less affected by the larger number of nodes that must be visited.

4.2.2 *Impact of range search workload on performance*

Figures 8, 9, 10, and 11 analyze the performance impact of doubling the query rate for each of the range search approaches, while keeping the reporting rate constant at 1 report every 5 minutes from each of the 900 virtual nodes in the overlay topology.

Figure 8 shows that the performance of the centralized approach decreases significantly when the query rate is doubled, due to an overloading of the links into the eight server nodes and increased CPU load on the eight servers. Note that because we are measuring a combined network and CPU effect, the performance degradation can be mitigated by increasing the number of well-connected servers from 8 to 64. The figure also shows that for the light query workload, a configuration with 8 servers actually slightly outperforms a configuration with 64 servers, due to the need to query and receive responses from a larger number of servers in the latter case.

Figure 9 shows that the performance of the hybrid approach deceases when the query rate is doubled, but it does not diminish as significantly as does the centralized approach. This is because index servers in the hybrid approach receive queries and rebroadcast queries, whereas servers in the centralized approach receive queries and return results. Because queries are significantly smaller than results, doubling the number of queries increases the traffic load on the index servers less than it increases the traffic load on the

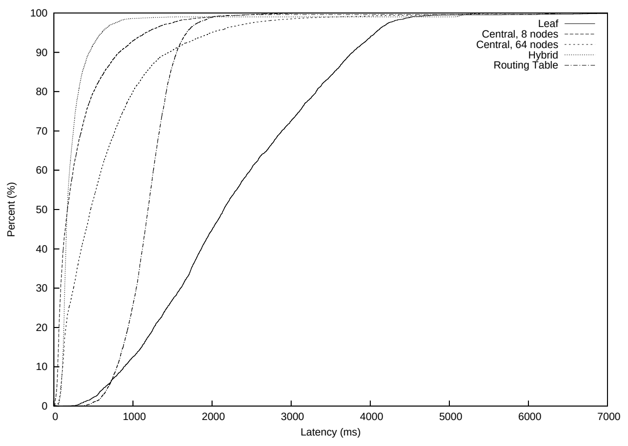


Figure 12: Cumulative distribution function of query latency for the four approaches under the default workload.

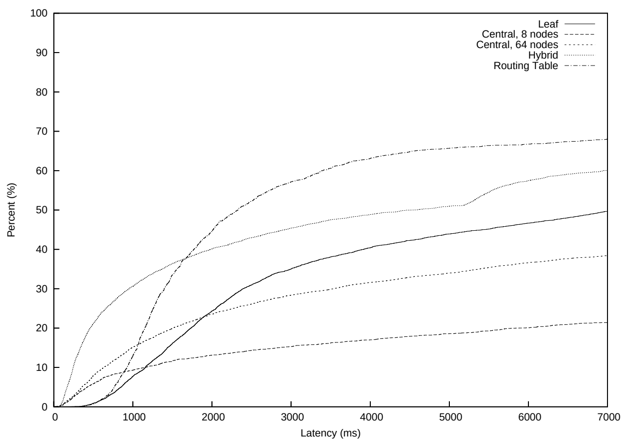


Figure 13: Cumulative distribution function of query latency for the four approaches under the 10x report workload.

central servers. (This is also why increasing the number of servers for the hybrid approach did not improve performance, while it did improve performance for the centralized approach.) The exact difference depends on the number of DHT nodes that the index server rebroadcasts the query to compared to the number of candidate nodes in the result set, and is therefore partly a function of the workload we have selected.

Figures 10 and 11 show that the performance of the leaf-set walk and routing-table walk approaches decreases very slightly with a doubling of the query rate. We were unable to run with larger query rates because of limitations of our test cluster. We do expect that as the query rate scales, the performance of DHT-based approaches will also eventually suffer.

Figures 12 and 13 analyze the performance impact of scaling up the report workload by 10x for each of the approaches, while keeping the query rate the same (one query every 5 minutes from each of 900 virtual nodes participating in the system). We do this by modeling what would happen if each SWORD node were acting as a proxy for ten non-SWORD nodes reporting statistics about themselves. Thus instead of each SWORD node sending a report about itself once every 5 minutes, as in the default workload, each SWORD node sends a report about *ten different nodes* every 5 minutes. This serves to increase both the number of reports sent per unit time and the size of query responses (because now up to 9000 nodes could match each query rather than 900 nodes).

We see the same trend in performance degradation when scaling up the report workload by 10x as we did when scaling up the query workload by 2x, except that the performance degradation for all four approaches is more pronounced when scaling up the report workload by 10x than when scaling up the query workload by 2x. This is because doubling the query rate essentially doubles the amount of network bandwidth used per unit time for every approach, while increasing the report workload by 10x increases by 10x the bandwidth used for reports and increases by approximately 10x the bandwidth used for query replies since there are 10x more nodes about which reports are issued.

We also observe that when scaling either the report workload or query workload, the performance of the leaf-set walk approach degrades more, in absolute terms, than does the performance of the routing-table walk approach. This is because the routing-table walk approach exploits more parallelism than does the leaf-set walk approach, so the increased latency along IP links is overlapped more in the routing-table walk than in the leaf-set walk. Also, the routing-table walk follows routing-table pointers, which are latency-optimized in Bamboo, while the leaf-set walk follows leaf set pointers, which are chosen purely by ordering nodes by their DHT ID, and therefore have expected latencies that are the average between any two overlay nodes in the DHT.

One conclusion from these results is that for a sufficiently large workload relative to the number of servers, a centralized implementation can perform poorly relative to a distributed one, depending on the amount of resources allocated to the centralized approach. Additionally, it may be easier for n organizations to each donate a single machine on behalf of a service rather than requiring a central service provider to provision many servers and the requisite network connections. We do not claim to have explored the full space of workloads and server configurations, so we are unable to make a general claim about the number of servers and quality of network connections to those servers necessary to outperform a distributed implementation. However, we do note that the DHT-based approached and a "hybrid" approach that distributes the data storage but centralizes the mapping of attribute ranges to data storage nodes, offer promising performance for at least some workloads.

4.2.3 Robustness to perturbations

It is often claimed that decentralized distributed hash tables are an important technology because the services built on top of them automatically inherit the DHT's performance scalability, self-configuration, and robustness. We have shown that SWORD inherits the first two properties in earlier sections; here we address the third. In contrast to centralized services that often retrofit reliability onto single-point-of-failure architectures (e.g., adding a failure-detecting front-end load balancer to a web service cluster), SWORD takes advantage of the Bamboo DHT's self-healing property that automatically remaps keys to nodes when a node

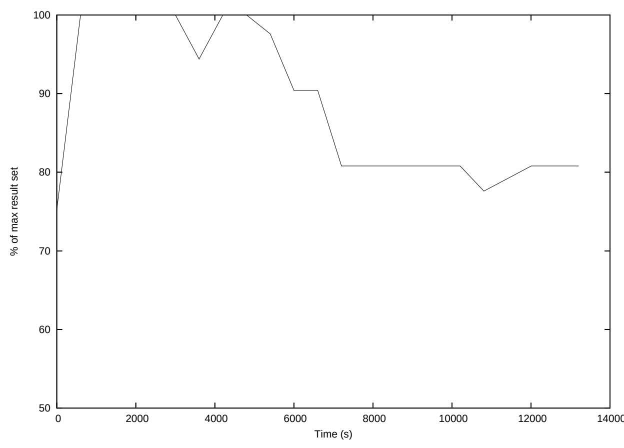


Figure 14: Number of candidate nodes returned by range query as a function of time, before and after killing 20% of the reporting nodes in a 900-node system at 5000 seconds into the run.

joins or leaves the system. As soon as the DHT has performed the necessary remapping, reporting nodes automatically send their reports to the node newly-responsible for the key corresponding to the report, and queries are automatically routed to the new nodes that will hold the relevant reports.

Each SWORD node periodically routes a measurement report to the DHT keys corresponding to the "key attributes" of the measurement report. If a DHT server node fails, information about the reporting node will therefore be unavailable until the next update, at which time the report will go to the DHT node that has taken over responsibility for that key. In other words, once the DHT stabilizes and a full reporting interval has passed, information about all live reporting nodes should be available to queries. To verify this robustness mechanism, we ran an experiment with the leaf-set walk DHT strategy, killing 20% of the DHT nodes approximately 5000 seconds into the run. The workload consisted of updates whose values were uniformly distributed between 0 and 1000, and queries for the range [0, 150] for one attribute. We computed over every ten-minute time interval the average of the number of nodes returned by the query, over all queries that were answered during that time interval. We expect this to be approximately $900 * 0.15 = 135$ at the beginning of the run, since approximately 15% of the 900 nodes will be reporting values in the range [0,150], and approximately $0.8 * 135 = 108$ after 80% of the nodes are killed. (Keep in mind that all nodes in the system are "servers" in the DHT storing measurement reports as well as load generators, so killing 20% of the nodes kills 20% of the reporting nodes as well as 20% of the servers.) Figure 14 shows this to be the case, by plotting the percentage of the maximum result set returned during each 10 minute interval. Once 20% of the nodes are killed at time 5000 seconds, SWORD and Bamboo "heal" and queries begin receiving the new result set (containing 80% of the original result set) within about 30 minutes. Between $t = 5000s$ and $t = 6800s$, queries are receiving information about some dead nodes because, for this run, we set the soft state timeout to 30 minutes (meaning that information about dead nodes is retained up to 30 minutes after they die). The minor fluctuations in the graph around $t = 3600s$ and $t = 10800s$ are due to occasional message loss due to network congestion.

4.2.4 Bandwidth Consumption

While range search latency and robustness are important, we are also interested in the relative network resource consumption of the four schemes. For the 10x report workload configuration (with 900 nodes each issuing measurement reports for 10 nodes every 5 minutes, and each of the 900 issuing one query every 5 minutes), we found the following average bandwidth consumption over the entire network for the four schemes, above the background bandwidth consumption of our network coordinates subsystem and Bamboo's leaf set and routing table membership maintenance processes. These numbers include bandwidth for sending measurement reports, performing range queries, and contacting the necessary "representative" for inter-node measurements.

The hybrid approach uses the most bandwidth because each DHT node must periodically beacon the lower and upper bounds of the DHT keyspace for which it is responsible, in addition to handling queries and reports. The routing-table walk and leaf approaches use comparable amounts of bandwidth; the difference is likely due to network congestion due to flooding in the routing-table approach leading to retransmissions. In theory the centralized approach should use the least bandwidth of all, since queries are rarely sent to servers that do not have possibly matching results. The unexpectedly large bandwidth consumption is due to a large number of UdpCC retransmissions due to high congestion on the servers' network links.

Although these bandwidth numbers are large, keep in mind that each of the 900 nodes in the system is sending reports about 10 nodes every 5 minutes, and issuing a query every 5 minutes. On PlanetLab, with slightly over 200 nodes and updates every few minutes containing over 40 attributes, we found network bandwidth consumption only about 5 Kbps above the baseline Bamboo bandwidth consumption (in the absence of queries), and this number could be further reduced with compression.

4.2.5 Load balancing

The Karger and Ruhl [15] load balancing mechanism we employ has the potential to disrupt the logarithmic-hops routing property of Bamboo because it shifts the distribution of nodes within the DHT ID space from uniform to more heavily concentrated in regions of high load. We were therefore interested in determining how significantly the load balancing process increases the average number of hops that messages take through the DHT. To evaluate this, we measured the average number of overlay hops taken by report messages on their way to the DHT node responsible for the report, every hour during a four-hour experiment with a Zipf workload where load balancing was enabled. We found that the average number of hops increased from 4.4 during the first hour to 6.2 during the fourth hour. This suggests that the node identifier skew does lead to a nontrivial violation of the desired logarithmic-hop routing property of the DHT, and that additional mechanisms may be needed to mitigate the effect if the routing inefficiency is considered unacceptable. Possible solutions to this problem are described in [4]. We are currently investigating ways to aug-

Range Search Technique	Average bandwidth
Hybrid	2.72 Mb/s
Centralized	2.45 Mb/s
Routing-table Walk	1.56 Mb/s
Leaf-Set Walk	1.43 Mb/s

Figure 15: Bandwidth per node, emulating 9000 reporting nodes/5 min and 900 queries/5 min.

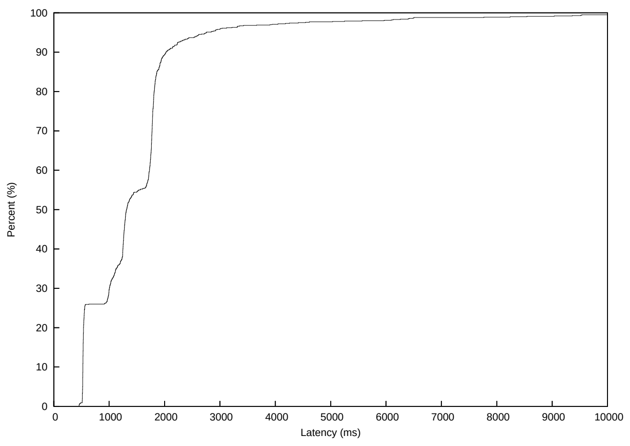


Figure 16: Cumulative distribution function of optimizer latency.

ment Bamboo with extra pointers to "route around" dense areas of the identifier space, as well as determining whether the potentially significant skew we have identified is significant in real-world workloads. For example, if update measurements (and hence loads) vary in a diurnal pattern, the paths with extra hops due to load balancing would change over the course of the day, serving to spread over time the impact of the routing length changes.

4.2.6 End-to-end performance

Range search performance is not the only contributor to end-to-end resource discovery query performance. We are also interested in the time to retrieve the inter-node statistics from "representatives" in parallel and for the optimizer to attempt to find the best mapping of nodes returned from the distributed query, to groups in the XML query specification.

For all runs in the 900 node configuration, the time needed to retrieve the inter-node statistics from "representatives" via TCP was approximately three seconds. Since there was one representative per stub, the maximum number of representatives that might need to be contacted was equal to the number stubs, which was 225. However, queries returned nodes with on the order of tens of candidate nodes, so the number of representatives that needed to be contacted was generally on the order of 10.

Figure 16 shows cumulative distribution function of completion times for 1000 queries passed to the optimizer. In order to create a large representative workload to evaluate the optimizer, resource updates were created using archived Ganglia [23] data gathered over three months on PlanetLab. Queries were chosen uniformly at random.

Putting together all the pieces, median end-to-end query latency (collecting single-node measurements via range query + collecting inter-node measurement by contacting representatives + running the optimizer) for a 900-node synthetic workload is less than ten seconds.

4.2.7 PlanetLab evaluation

In addition to evaluating SWORD in an emulated environment, we have evaluated SWORD's real-world performance on PlanetLab. Compared to our ModelNet configuration, PlanetLab has a smaller number of nodes, a higher per-node CPU load, and a wide range of inter-node bandwidths that depends on the nodes' location.

We ran our experiments on PlanetLab on July 16, 2004. We ran our experiments on two sets of nodes, one a subset of the other. The first set was all 214 usable nodes that were connected to the commodity Internet (i.e. all usable nodes that were not connected only to Internet-2)[1]. The second set of nodes was the subset of the first set that is used by CoDeeN. These 108 nodes are all at universities in North America and tend to have high-bandwidth, low-latency network paths to one another.

Each node reported 44 metrics, of which 31 were declared as "keys" (usable as the range search attribute in a query) and 13 were declared as "non-keys." We ran the experiments with updates at a 2 minute interval and at a 4 minute interval but found no significant difference in the performance results, so we report only the 2-minute interval results here. We measured query latency when a single query was in the system at a time; the measured times thus represent the "best case" latency. Queries were introduced into the system and returned to the user using a command-line client as depicted in Figure 1.

For our experiments we issued a series of queries of the form

```
<request>
   <query_attr>load_one</query_attr>
   <group>
      <name>Group1</name>
      <numhosts>10</numhosts>
      <num_machines>4</num_machines>
      <load_one>0.0,0.0,N,N,0.0</load_one>
      <latency>0.0,0.0,150.0,150.0,0.0</latency>
   </group>
   <group>
      <name>Group2</name>
      <numhosts>10</numhosts>
      <num_machines>4</num_machines>
      <load_one>0.0,0.0,N,N,0.0</load_one>
```

[1]Bamboo needs symmetric reachability among nodes, but a host that is connected only to Internet-2 cannot route via IP to a host that is not connected to Internet-2, and vice-versa; thus Bamboo cannot usefully run on nodes that are only connected to Internet-2, and by implication SWORD cannot run on nodes that are only connected to Internet-2.

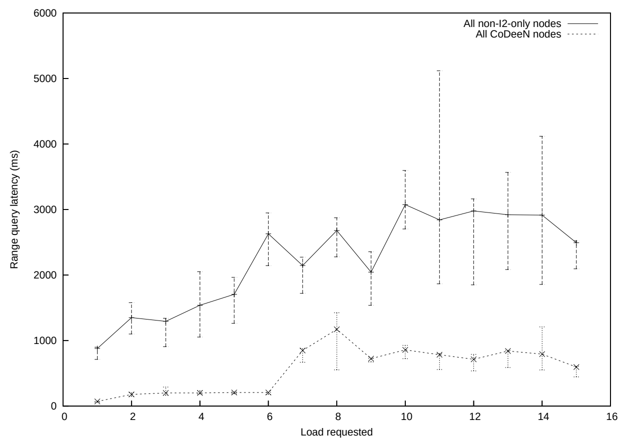


Figure 18: Range query latency versus width of range searched, on PlanetLab.

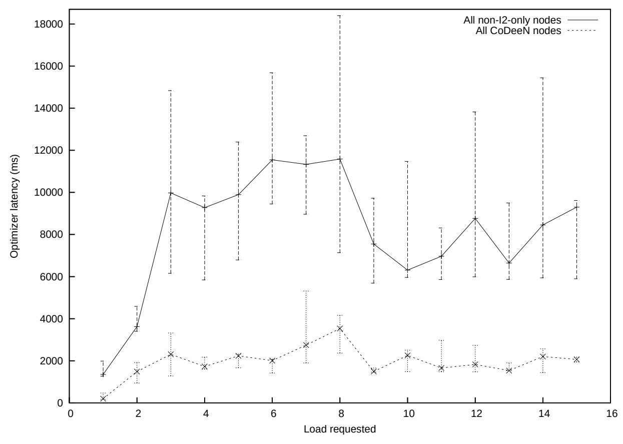


Figure 19: Optimizer latency versus width of range searched, on PlanetLab.

```
        <latency>0.0,0.0,150.0,150.0,0.0</latency>
    </group>
</request>
```

N was varied between each run, covering all integers between 1 and 15 inclusive. The f_A function for `load_one` was configured to map the range 0 to 15 to the full sub-region corresponding to the load_one attribute (32 sub-regions were used, hence approximately 1/32 of the nodes were mapped to the range for each attribute), with loads greater than 15 mapped to the same DHT key as loads equal to 15. With this configuration we thus expect approximately $(M/32)/15$ additional DHT nodes to be searched for each 1.0 increment in `load_one`, where M is the total number of nodes in the system.

Figures 18 and 19 shows the median latency for the distributed range query and the median latency to run the optimizer, as a function of the upper bound of the load requested (and hence the number of candidate nodes returned). For reference, the number of candidate nodes returned and the number of DHT nodes visited by each range query is listed in Figure 17. Note that in Figure 17 the number of DHT nodes visited increases unevenly as the "max load" increases by each 1.0 unit due to the relatively small number of nodes leading to a not-perfectly-uniform random distribution of node IDs in the DHT keyspace.

Figure 18 shows that SWORD's range search performs reasonably well on PlanetLab, returning results to the optimizer within a few seconds even when all nodes are returned by the range query. The graph shows that most performance effects are in the "noise" except for the number of candidate nodes returned: the load 1.0 response time for the "all-nodes" configuration is about the same as the load 15.0 response time for the CoDeeN configuration, even though in the latter configuration three times as many DHT nodes are visited, because in both cases 108 candidate nodes are returned. As further evidence, increasing the number of DHT nodes visited does not increase range query latency for either configuration as the number of candidate nodes returned remains approximately equal (e.g. in the load 6.0 to 15.0 regimes). This suggests that *if* real-world user queries commonly return hundreds of candidate nodes, we can improve SWORD's performance by reducing the amount of data transferred from DHT server nodes to the querying node by encouraging users to explicitly limit the number of candidate nodes returned in their query as described earlier, and/or by using compression of data about returned candidate nodes.

SWORD's optimizer latency (Figure 19) shows a similar effect of latency increasing as the number of candidate nodes increases. Here the increasing latency is not due to the larger amount of data transferred, but because of the larger input data set to the optimizer algorithm. This graph shows that optimizer latency can be significant if a large number of candidate nodes is returned, and that we should attempt to reduce the running time of the optimizer in future work. Note that some performance degradation effects are magnified in this graph because of high load on all PlanetLab nodes, and therefore on the PlanetLab node issuing the query and running the optimizer.

5. RELATED WORK

The existing work most closely related to SWORD falls into three categories: systems for distributed range search, systems for internet-scale query processing (that do not support distributed range search), and systems for wide-area node resource discovery.

The three related systems for distributed range search are Karger and Ruhl, Mercury, and PHT. The general idea of mapping values of stored items to the DHT node responsible for the key with that value, in order to enable range search, is alluded to briefly by Karger and Ruhl [15]. In contrast to our work, they did not evaluate this idea (either in implementation or simulation), and they did not suggest the mechanisms we use for "passive" load balancing or any technique for mapping to the DHT keyspace the values of attributes whose values do not naturally fall into the DHT keyspace.

Perhaps most closely related to our work is a concurrent effort to support distributed range queries in Mercury [4]. Essentially, Mercury adopts a strategy similar to our "leaf set walk" approach. Mercury uses a "small-worlds" DHT structure that mitigates the path length changes due to Karger and Ruhl-style load balancing, whereas SWORD runs unmodified on an existing DHT (Bamboo). Their motivating application is object management for a multiplayer

Max load	all-nodes # candidate nodes returned	all-nodes # DHT nodes visited	CoDeeN-nodes # candidate nodes returned	CoDeeN-nodes # DHT nodes visited
1.0	108	2	34	1
2.0	167	2	80	1
3.0	195	3	99	1
4.0	203	3	99	1
5.0	204	3	99	1
6.0	205	3	105	1
7.0	206	3	106	2
8.0	208	4	107	3
9.0	209	4	107	3
10.0	213	6	107	3
11.0	213	7	107	4
12.0	213	7	107	6
13.0	213	8	107	6
14.0	214	9	108	6
15.0	214	9	108	6

Figure 17: Number of candidate nodes returned and number of DHT nodes visited, as a function of load, for the "all non-I2-only nodes" and "all CoDeeN nodes" configurations reported in Figures 18 and 19.

game, whereas we target wide-area resource discovery. As a result, we have a separate mechanism for retrieving large objects ("representatives"), we handle network coordinates and inter-node constraints, and we support a query language with user-specified utility functions. Finally, we evaluate three peer-to-peer DHT based techniques and compare them to a centralized implementation.

Another recent mechanism proposed for distributed range search is the Prefix Hash Tree (PHT). PHT incrementally grows a trie, each of whose leaves corresponds to a range of item identifiers that map to the DHT node responsible for the DHT key defined by the path from the root of the trie to that leaf. The trie starts with a singleton node (the root) and grows by incrementally splitting nodes when they become overloaded. Leaf nodes are merged into their parent (which becomes a leaf) when the children become underloaded. Although we considered using PHT for range search, we decided against it for several reasons. First, PHT defines overload and underload as absolute conditions rather than relative to other nodes; thus is can prevent overload and underload but would be expected to be less effective at *balancing* load because no comparison is made of load among nodes. Second, the base PHT design requires maintaining a small amount of extra metadata beyond what is required by the DHT (in particular, each node needs to know whether it is an interior node or a leaf), and the important performance optimizations suggested for PHT require additional metadata beyond that. The Karger and Ruhl approach (and therefore ours) requires no extra metadata beyond that which the DHT is already maintaining for the purposes of key-based routing. Note also that because PHT maintains one trie per attribute, the amount of metadata maintained will be multiplied by a nontrivial constant factor in a system such as ours where many attributes are tracked. The amount of PHT metadata is not as much of a concern because of the associated storage requirements, which is small, as because it must be kept up-to-date in the face of nodes joining and departing the network, voluntarily or due to failures and recovery. A pure DHT-based approach such as ours leverages the dynamic metadata maintenance provided by the DHT.

Three systems for internet-scale query processing are PIER [13], Sophia [28], and IrisNet [17]. Although all three could be used for resource discovery, they do not support distributed range search, so range search queries are expected to perform more poorly than in SWORD. In particular, a range query would be flooded to all nodes and the range filter applied on the node that issued the query. Also, they do not offer special support for resource discovery queries, such as SWORD's XML-based query language, its support for constraints on inter-node attributes, or its utility-based optimization framework. A related system, Ganglia [23], collects metrics from cluster nodes, and has recently added functionality to hierarchically aggregate this data for presentation at a centralized node, but it is not fully distributed in SWORD's peer-to-peer fashion, and it lacks a query language. As mentioned in Section 3.6, we use Ganglia's per-node daemon as one of our data sources for our SWORD service on PlanetLab.

Researchers have proposed a number of systems or approaches for node resource discovery in wide-area systems. Considine, Byers, and Mayer-Patel [6] argue that next generation wide-area testbeds should possess the same specifiable and repeatable behavior that is present in emulation and simulation environments. They go on to describe a constraint satisfaction method for finding a topology of nodes that meets a set of pairwise constraints, and they note that this problem is NP Complete. The authors are essentially solving the same group creation problem based on inter-node characteristics that we are, but they do not handle single-node measurements, and their tool does not offer a query language or service interface.

SWORD bears some resemblance to XenoSearch [24]; like XenoSearch, we partition attribute value ranges among nodes in a DHT. However, we allow value ranges to be partitioned unequally among nodes to compensate for uneven distributions of the source data. We also allow partitioning on arbitrary attributes and "server"-side filtering of the other attributes, as opposed to XenoSearch which partitions all attributes and finds nodes matching desired characteristics by intersecting the sets of nodes that match each attribute constraint individually. Thus, they trade higher query overhead for potentially lower update overhead since each DHT update message only contains a single value rather than all values as in SWORD. XenoSearch also uses a separate DHT overlay per attribute, with the attendant overhead. Finally, SWORD queries can specify constraints on inter-node characteristics, while XenoSearch allows searches only over single-node characteristics, and XenoSearch returns all nodes matching the user-specified constraints rather than returning a utility-optimized number of nodes the user has requested.

Grid technologies [8, 29] address the issue of resource discovery as part of the Globus Grid toolkit. The Grid views resources contributed by companies and universities as virtual organizations (VO). MDS-3 supports resource discovery in Globus. The MDS-3 architecture consists of two entities: information providers and aggregate directories. Information providers contain detailed information about Grid entities. When combined, information provider services form a VO-neutral infrastructure that provides access to information about Grid entities. Aggregate directories provide specialized, VO-specific views of federated resources. They store information from information providers, and respond to queries using the stored information. MDS-3 uses two base protocols, GRIP and GRRP. The Grid Information Protocol (GRIP) accesses information from information providers in the Grid. The Grid Registration Protocol (GRRP) notifies aggregate directory services that information is available. MDS-3 also includes a configurable information provider framework called a Grid Resource Information Service (GRIS), which can be customized by plugging in specific information sources. Current information source implementations include static and dynamic host information, as well as network information. GRIS parses GRIP requests and dispatches them to the appropriate information provider, which returns the results back to the client. We are not aware of any detailed evaluation of this architecture, such as its performance, scalability, or availability. However, we believe that our architecture and techniques in SWORD are orthogonal to these important efforts and could be integrated into the MDS-3 infrastructure. One advantage SWORD does provide over Globus and MDS-3 is a query language that allows for queries over inter-node attributes. Queries of the type described in this paper are difficult to express in the standard query language of MDS-3.

In [1], the authors present the design and implementation of INS, an Intentional Naming System. Using the scheme they propose, applications that use INS specify what they are looking for in the network, rather than specifying a specific location or hostname that describes where to find the needed resource. INS includes a simple language based on attributes and values. To satisfy requests, INS request resolvers form an application level overlay network that discovers and monitors new services and resources. This system provides a way to locate services and resources based on resolver-initiated advertisements. While this initial work was largely restricted to local area networks and faced some scalability limitations, more recent work on Twine [2] extends the Twine architecture to wide-area settings, by partitioning resource description among nodes participating a Chord-based overlay network. Relative to our work on SWORD, Twine targets a different set of applications and, for instance, does not focus on either range searches or or inter-node characteristics.

Huang and Steenkiste [12] describe a mechanism for network-sensitive service selection. Their system addresses a problem similar to the one we describe here, but using centralized data collection and resource mapping. They also focus on finding single groups that meet target criteria for a desired application, rather than multiple groups with specific inter-node and inter-group characteristics.

6. CONCLUSIONS

A key infrastructural requirement for emerging federated large-scale computation and communication environments is a resource discovery system that allows users to locate subsets of global resources to host their applications. The service must be flexible enough to support the requirements of a broad range of applications, highly available as it serves as the entry point for service deployment, and scalable both to very large systems and to a large number of rapidly changing monitored characteristics.

To this end, this paper explores a variety of architectural alternatives for such a service, ranging from centralized to fully distributed, through a detailed performance evaluation under realistic wide-area network conditions and for a variety of workloads. In determining the appropriate system structure and abstractions, we make the following novel contributions. First, we provide support for efficient distributed range queries over the conjunction of per-node resource requirements. Second, we efficiently support queries over inter-node characteristics such as latency and bandwidth in addition to the more traditional set of single-node metrics. Third, we allow users to issue queries containing simple “utility functions” and we develop an optimizer that approximately maximizes the utility of the nodes returned from the query. Finally, we use passive and active load balancing schemes for partitioning node attributes across distributed system participants and for dynamically adapting to skewed resource popularity. From evaluating SWORD in both an emulated environment and a real-world deployment on PlanetLab, we conclude that the combination of techniques SWORD uses provide sufficient performance and robustness to serve as a prototype “production” resource discovery service. In so doing, we also provide evidence for the claims many have made that decentralized distributed hash tables are a useful building block for developing scalable, highly available, self-configuring wide-area distributed services.

7. REFERENCES

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The Design and Implementation of an Intentional Naming System. In *Symposium on Operating Systems Principles*, December 1999.

[2] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *International Conference on Pervasive Computing*, August 2002.

[3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating Systems Support for Planetary-Scale Network Services. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, March 2004.

[4] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. Technical report, Carnegie Mellon University, January 2004. Revised version to appear in ACM SIGCOMM 2004.

[5] H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *Proceedings of ACM SIGMETRICS*, June 2002.

[6] J. Considine, J. Byers, and K. Mayer-Patel. A Constraint Satisfication Approach to Testbed Embedding Services. In *Proceedings of ACM HotNets-II*, November 2003.

[7] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris. Practical Distributed Network Coordinates. In *Proceedings of the ACM HotNets Workshop*, November 2003.

[8] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing, 2001.

[9] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002.

[10] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal on Supercomputer Applications*, 15(3), 2001.

[11] K. Hildrum, J. D. Kubiatowicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, August 2002.

[12] A.-C. Huang and P. Steenkiste. Network-sensitive service discovery. In *The Fourth USENIX Symposium on Internet Technologies and Systems*, 2003.

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

[14] A. Iamnitchi and I. Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *Proceedings of the International Workshop on Grid Computing*, November 2001.

[15] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proceedings of the International Peer to Peer Symposium (IPTPS)*, February 2004.

[16] B. Krishnamurthy and J. Wang. On Network-Aware Clustering of Web Clients. In *Proceedings of ACM SIGCOMM 2000*, August 2000.

[17] S. Nath, Y. Ke, P. B. Gibbons, B. Karp, and S. Seshan. IrisNet: An Architecture for Enabling Sensor-Enriched Internet Services. Technical Report IRP-TR-03-04, Intel Research Pittsburgh, June 2003.

[18] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *Proceedings of INFOCOM*, June 2002.

[19] T. S. E. Ng and H. Zhang. A Network Positioning System for the Internet. In *USENIX Annual Technical Conference*, June 2004.

[20] V. Pai. CoTop: A Slice-Based Top for PlanetLab. http://codeen.cs.princeton.edu/cotop/.

[21] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht, 2004.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware'2001*, November 2001.

[23] F. D. Sacerdoti, M. J. Katz, M. L. Massie, and D. E. Culler. Wide Area Cluster Monitoring with Ganglia. In *Proceedings of the IEEE Cluster 2003 Conference*, December 2003.

[24] D. Spence and T. Harris. Xenosearch: Distributed resource discovery in the xenoserver open platform. In *Proceedings of HPDC*, 2003.

[25] 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 the 2001 SIGCOMM*, August 2001.

[26] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-scale Network Emulator. Technical report, Duke University, May 2002.

[27] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. 21(2):164–206, May 2003.

[28] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *Proceedings of ACM HotNets-II*, November 2003.

[29] X. Zhang, J. L. Freschl, and J. M. Schopf. A performance study of monitoring and information services for distributed systems. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 270. IEEE Computer Society, 2003.