Design and Implementation Tradeoffs for Wide-Area Resource Discovery

JEANNIE ALBRECHT
Williams College
and
DAVID OPPENHEIMER
Google Inc.
and
AMIN VAHDAT
University of California, San Diego
and
DAVID A. PATTERSON
University of California, Berkeley

We describe the design and implementation of SWORD, a scalable resource discovery service for wide-area distributed systems. In contrast to previous systems, SWORD allows users to describe desired resources as a topology of interconnected groups with required intra-group, inter-group, and per-node characteristics, along with the utility that the application derives from specified ranges of metric values. This design gives users the flexibility to find geographically distributed resources for applications that are sensitive to both node and network characteristics, and allows the system to rank acceptable configurations based on their quality for that application.

Rather than evaluating a single implementation of SWORD, we explore a variety of architectural designs that deliver the required functionality in a scalable and highly-available manner. We discuss the tradeoffs of using a centralized architecture as compared to a fully decentralized design to perform wide-area resource discovery. To summarize our results, we found that a centralized architecture based on 4-node server cluster sites at network peering facilities outperforms a decentralized DHT-based resource discovery infrastructure with respect to query latency for all but the smallest number of sites. However, although a centralized architecture shows significant promise in stable environments, we find that our decentralized implementation has acceptable performance and also benefits from the DHT's self-healing properties in more volatile environments. We evaluate the advantages and disadvantages of centralized and distributed resource discovery architectures on 1000 hosts in emulation and on approximately 200 PlanetLab nodes spread across the Internet.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Design, Experimentation, Performance, Reliability

Additional Key Words and Phrases: Resource Discovery, PlanetLab

---

---

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 [Bavier et al. 2004] and the Grid [Foster et al. 2001] have become popular environments for evaluating and deploying such services. These platforms are typically comprised of large numbers of heterogeneous machines connected to the Internet behind links of varying bandwidth and latency. One difficulty in the practical use of these large-scale infrastructures centers around locating an appropriate subset of system resources to host a service, computation, or experiment. The process of locating a set of machines to run a distributed application is often called *resource discovery*. This paper explores the design decisions that must be made when building a highly available and user-friendly service that performs resource discovery for users who wish to run applications in wide-area, heterogeneous environments, specifically focusing on PlanetLab.

The principal challenges to performing resource discovery in federated Internet systems include the heterogeneity of the underlying resources, dynamically changing per-node characteristics such as CPU load and free memory, and application sensitivity to inter-node characteristics such as bandwidth and latency. Each distributed application also has a different set of resource requirements, which means that the best set of resources will vary for each application. For example, compute-intensive applications, such as parallel scientific applications, might be particularly concerned about available CPU, physical memory, and disk capacity on machines (or nodes) hosting the application. 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 machines with low-latency, high-bandwidth links among themselves. Other applications, such as multiplayer games, might be concerned about per-node CPU resources and inter-node latency and bandwidth. Additional characteristics may be of interest to all applications: deploying on nodes with high historical availability may reduce performance degradations due to failures, while deploying on nodes with low resource variability over time may improve service predictability. Thus, a resource discovery infrastructure designed to support a variety of distributed applications must accurately track per-node and inter-node characteristics and provide support for a range of semantics with respect to specifying resource requirements.

In addition to supporting a variety of application resource requirements, there are a number of other factors to consider when designing a resource discovery service for use in distributed environments. To help guide our design decisions and before discussing the architecture of a specific resource discovery service, we extract the following key system requirements and goals. First, to be useful for finding resources in large-scale infrastructure-based platforms [Bavier et al. 2004; Foster et al. 2001] and end-user-based platforms [Red Herring Magazine 2004], the system must *scale* to thousands of nodes and sites. Second, it must be *highly available*, as it is the entry point into the system for service deployers wishing to find nodes to host their application. Third, since certain node characteristics vary rapidly, the system must support *high rates of measurement updates* from participating machines. Fourth, the system must track both *static* characteristics such as operating system, processor speed, and network coordinates [Dabek et al. 2004; Ng and Zhang 2002; 2004], as well as more *dynamic* characteristics such as available CPU, memory, and disk resources. Fifth, the system must support queries over not just per-node characteristics such as load and network location, but also over *inter-node characteristics* such as latency and bandwidth. Lastly, due to the fact that different applications have widely varying needs and place varying priorities on those needs, the system must offer an expressive query language that allows specifying ranges of required resource quantities, as well as information about how much *utility* is lost from the selection of imperfect but acceptable nodes. Thus the resource discovery service we envision combines aspects of distributed measurement, distributed query processing, and user utility specification and optimization.

A number of recent efforts have explored large-scale resource discovery [Balazinska et al. 2002; Huang and Steenkiste 2003; Czajkowski et al. 2001; Spence and Harris 2003; Wawrzoniak et al. 2003; van Renesse et al. 2003]. However, to our knowledge no existing system meets all of the above requirements. In particular, we believe that resource discovery systems must support specifying required inter-node characteristics and the relative utility of both per-node and inter-node characteristics. Further, the system must consider the overall utility of a group of resources when making decisions and assessing the tradeoffs among competing potential configurations. One contribution of this work is to present the query semantics of SWORD, a resource discovery infrastructure that allows users to easily describe desired resources as a *topology of interconnected groups* with required intra-group and inter-group characteristics and *penalty functions* to indicate the utility of those characteristics to the application. While we cannot prove generality and indeed there are certain semantics that we cannot capture, we provide examples of a set of disparate applications that we map to SWORD's semantics.

A resource discovery system meeting our requirements has the potential to generate significant load, both in terms of monitoring the target infrastructure, and in terms of running the NP-hard group finding algorithm required to answer queries for resources described as interconnected groups. Gathering, storing, and querying the monitoring data requires sufficient storage, network, and CPU resources to support the measurement update and query rate of the system. The group-finding algorithm, which is analogous to the classic *k-clique* complexity problem, requires substantial CPU resources for each query. The generated load of a resource discovery infrastructure increases dramatically as the number of users submitting queries and available resources increase, in turn impacting system scalability and performance. Thus, the second contribution of this work is an exploration of a variety of architectures to support SWORD's design.

In this paper, we consider building SWORD using three different architectures, and quantify the advantages and disadvantages of each design. Our first approach is fully distributed, and is based on the intuition that some type of decentralized architecture is required for scalability, load balancing, and fault tolerance (similar to a variety of earlier related efforts [Balazinska et al. 2002; Huang and Steenkiste 2003; Spence and Harris 2003; van Renesse et al. 2003]). One of the key disadvan-

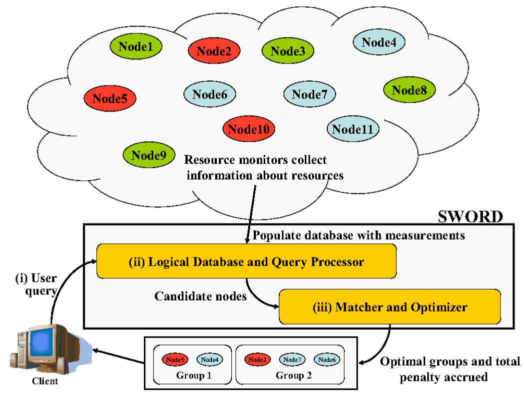


Fig. 1. High-level architecture of SWORD.

tages to any distributed architecture when compared to its centralized counterpart is the complexity required to maintain consistency among distributed participants, which often impacts performance. To quantify the overhead caused by this added complexity in a resource discovery system, our second architecture is fully centralized. We show that while the distributed architecture offers some advantages with respect to scalability and fault tolerance, under certain operating conditions a centralized approach outperforms its distributed counterpart. Additionally, we find that a third "hybrid" design, which combines aspects of both centralized and distributed solutions, has the potential to outperform both techniques.

The remainder of this paper is organized as follows. Section 2 presents a high-level overview of the SWORD architecture and Section 3 describes the details of our implementation. We describe our evaluation infrastructure and performance results in Section 4, and in Section 5 we describe qualitative lessons learned from operating SWORD as a PlanetLab service for the past two years. We present related work in Section 6 and conclude in Section 7.

2. SYSTEM ARCHITECTURE AND QUERY REPRESENTATION

This section explains our assumptions and terminology, describes SWORD's high-level architecture, and describes the query model that drives our design.

2.1 Architectural Overview

SWORD's architecture consists of three parts, illustrated schematically in Figure 1. The first component, labeled (i), is a *query syntax* for specifying desired node characteristics. The SWORD query syntax is described in detail in Section 2.4. The second component, labeled (ii), is a distributed *query processor* for finding candidate nodes whose characteristics match the specified requirements. Logically, this is analogous to a database and a query engine. In the case of SWORD, having a single centralized database does not scale well to high amounts of users and resources[1]. In Section 3.1, we explore other approaches that leverage the scalability provided by distributed hash tables. The third and final component of SWORD, labeled (iii),

[1]We use the terms resources, nodes, hosts, and machines interchangeably.

is the *optimizer* that finds a utility-maximizing mapping of a subset of the candidate nodes returned by the query processor to the groups in the user's queries, accounting for desired per-node, intra-group (or inter-node), and inter-group characteristics. The optimizer outputs this mapping, along with the measurements that led to those nodes' selection. Section 3.2 presents the design of the optimizer.

SWORD can return many such mappings if desired, ranked from lowest to highest penalty, but for clarity we describe SWORD as if it outputs only the lowest-penalty configuration. Note that while we often refer to the "client" or "user" as a human submitting a query, in reality, the "user" may be an application such as an execution management system [Albrecht et al. 2006] who is submitting a query on behalf of a human user. This distinction does not affect the operation of SWORD.

Note that an additional (and unlabeled) component in Figure 1 is the set of resource monitors used to populate the logical database. SWORD does not dictate the mechanisms for measuring per-node and inter-node attributes, so we do not discuss such mechanisms here in detail. In Section 3.3 we describe our deployment of SWORD on PlanetLab and mention the specific resource monitors that SWORD uses in that deployment.

2.2 Target Usage Scenarios

One of the key assumptions in our development of SWORD is that application deployers have some incentive to avoid deploying their application on all nodes available to them. This incentive may be altruistic, financial, performance-related, or scientific (e.g., to evaluate an application's performance and robustness under various configurations). Hence, SWORD is designed with two usage scenarios in mind. When used in a "best effort" environment such as PlanetLab, SWORD matches a user's specification of desired resource characteristics to the resource characteristics of nodes at the time the request is made. SWORD returns an ordered list of sets of nodes, ranked by the closeness of each set's match to the user's desires. The substantial heterogeneity in available node resources and network characteristics across both space (nodes and network links) and time on shared platforms such as PlanetLab is quantified elsewhere [Oppenheimer et al. 2006; Rhea et al. 2005]; SWORD helps users cope with this heterogeneity when they deploy and operate their distributed applications.

In the second usage scenario, SWORD operates in conjunction with an external resource allocation or admission control mechanism such as might be the case with Grid systems [Foster and Kesselman 2003]. Here, the resource allocation system might augment the measurement database with information about which resources each user can access, at what cost, and during what time periods. Queries would then specify a desired period of time for using the requested resources, and candidate nodes would be filtered to exclude nodes unavailable to a user for the requested period. Additionally, the financial cost of each returned configuration is indicated as a function of the total utility of the resources. Although our deployment of SWORD thus far has been in a "best effort" environment, we expect to integrate it with resource allocation tools such as SHARP [Fu et al. 2003] or SNAP [Czajkowski et al. 2002] in the future to support arbitrated usage scenarios.

In both usage scenarios described above, SWORD assumes that users consult an out-of-band mechanism for determining what the valid query attributes are for the target infrastructure. SWORD places few restrictions on the format and type of attributes available to the user. We discuss the acceptable formats in detail in Section 2.4. SWORD supports queries for any attribute in the logical database, which includes any values that the target infrastructure has a mechanism for measuring and reporting. Some example attributes are available operating systems, per-node measurements such as load and free memory, inter-node measurements such as latency and bandwidth, information about node firewall status, and disk protection requirements. The live deployment of SWORD on PlanetLab uses a webpage that lists all available attributes.

2.3 Query Semantics

Two observations guide the design of SWORD's query representation. First, we observe that it is common for distributed services to be composed of groups of nodes that cooperate closely within a group, and, in some cases, more loosely among groups. One simple example is a "client-server" web application, where the servers are a group of well-connected high bandwidth, powerful machines, and the clients are less powerful, geographically distributed machines. A survey of common Grid applications described in [Kee et al. 2005] mentions that some scientific applications desire collections of tightly-coupled groups of nodes. In addition to these examples, consider the following more specific scenarios:

— An Internet search engine consists of a dozen sites distributed worldwide that are close (in network topology) to various large user populations. Each site is comprised of of a number of nodes which together hold a full copy of the search index. Nodes within a site coordinate in an all-to-all fashion to implement techniques such as parallelized index searching and cooperative caching. Among sites, a small number of nodes communicate periodically to share newly-crawled data. This service desires sufficient per-node storage, low-latency and high-bandwidth network links among nodes in each site, and at least one high-bandwidth link to connect each pair of sites.

— A data-intensive scientific application utilizes data stored in several geographically distributed data warehouses. The application needs a few nodes near each data source with low-latency and high-bandwidth connections to the data source to perform filtering and summarization of data from that source, and a large number of powerful compute nodes with low-latency, high-bandwidth connections among one another to perform a distributed computation over the data received from the nodes that perform filtering and summarization.

— A Content Distribution Network such as CoDeeN [Pai et al. 2003] wishes to place instances of its service on nodes near each of several geographically distributed user populations, with low-latency, high-bandwidth links among the nodes for efficient transfer of content between forward and reverse proxies. Moreover, to support high workloads, the application deployer wants each instance to be not a single node but rather a "virtual cluster" of several machines, all nearby in network topology and with low latency, high bandwidth links among them.

With these examples as motivation, resource specifications in SWORD focus on the notion of *groups* that capture equivalence classes of nodes. Each group consists of nodes with similar per-node and inter-node characteristics, as well as constraints between pairs of nodes in different groups.

The second observation guiding the design of SWORD's query semantics is that applications have varying sensitivities to deviations from specified per-node and inter-node characteristics. Thus, rather than specifying a single acceptable value for the amount of free memory desired, for example, we give users the flexibility to specify a range of acceptable values for free memory, and assign a utility to this range. To achieve this, users specify absolute requirements on per-node and inter-node characteristics, stricter preferred per-node and inter-node characteristics, and the sensitivity of their application to deviations from the preferred values. With this method, nodes whose value for an attribute fall within the absolute required range are ranked by their suitability based on their deviation from the preferred range. The user describes this sensitivity using per-attribute *penalty functions*, which can be thought of as the inverse of utility functions. The penalty function is designed to allow the user to express the application's sensitivity to each attribute and the relative importance of preferences for different attributes.

For example, consider the small Internet search engine, with large user populations in North America and Europe. The service operator requests two groups of nodes, one in each of those geographic regions. With respect to per-node storage, the operator requests that all machines selected for the Europe group have at least 1000 MB of free disk space, but that under constraint, machines with at least 300 MB of free disk space are acceptable. Any candidate machine with between 300 and 1000 MB of free disk space is assigned a penalty proportional to the deviation from 1000 MB. Machines with more than 1000 MB of free disk space are assigned a zero penalty, and machines with less than 300 MB of free disk space are discarded. In this example, the acceptable range of values for free memory is [300 MB, infinity]. The preferred range of values in this example, which is always a subset of the acceptable range, is [1000 MB, infinity]. SWORD responds to this query with a group of nodes that are the lowest-penalty configuration—defined as the sum over all groups of each member node's penalty, taking into account per-node, inter-node, and inter-group constraints—that meets all of the resource requirements specified in the query.

2.4 Expressing Queries

SWORD offers two query syntaxes: a "native" XML syntax and the ClassAds syntax [Raman et al. 1998] used by the Condor workload management tool. Because the standard ClassAds syntax is somewhat limited and cannot be used to express inter-node properties, SWORD users can only constrain and rank configurations based on per-node properties when using the ClassAds syntax. When evaluating queries expressed using ClassAds syntax, we invoke the ClassAds evaluator rather than our optimizer. The ClassAds evaluator is computationally simpler than SWORD's native optimizer because the ClassAds evaluator only considers per-node properties; ranking a configuration is therefore linear in the number of nodes in the configuration. In contrast, inter-node and inter-group constraints make SWORD's optimization problem exponential in the number of nodes.

The native SWORD XML syntax for a sample query that might be issued by the search engine operator appears in Figure 2(a), with the corresponding response

```
RequeryInterval 60
Group NA
 NumMachines 4
 Required Load [0.0, 2.0]
 Preferred Load [0.0, 1.0], penalty 100.0
 Required FreeDisk [500.0, MAX]  (MB)
 Preferred FreeDisk [1000.0, MAX], penalty 0.2
 Required OS [``Linux'']
 Required AllPairs Latency [0.0, 20.0] (ms)
 Preferred AllPairs Latency [0.0, 10.0], penalty 2.0
 Required AllPairs BW [0.5, MAX] (Mb/s)
 Preferred AllPairs BW [1.0, MAX], penalty 2.0
 Required Location [``NorthAmerica'', 0.0, 50.0] (ms)
Group Europe
 NumMachines 4
 Required Load [0.0, 2.0]
 Preferred Load [0.0, 1.0], penalty 100.0
 Required FreeDisk [300.0, MAX]  (MB)
 Preferred FreeDisk [1000.0, MAX], penalty 100.0
 Required OS [``Linux'']
 Required AllPairs Latency [0.0, 20.0] (ms)
 Preferred AllPairs Latency [0.0, 10.0], penalty 2.0
 Required AllPairs BW [0.5, MAX] (Mb/s)
 Preferred AllPairs BW [1.0, MAX], penalty 2.0
 Required Location [``Europe'', 0.0, 50.0] (ms)
InterGroup
 Required OnePair BW NA Europe [3.0, MAX] (Mb/s)
 Preferred OnePair BW NA Europe [5.0, MAX], penalty 0.5
```

(a)

```
SWORD Groups
-------------------
  Name: Group NA
  Max Latency: 20.0 ms
  Min Latency: 0.0 ms
  Overall Group Cost: 4.1
  Group Size:  4
    planetlab6.nbgisp.com
    planet1.halifax.org
    planet02.csc.ncsu.edu
    planetlab1.kscy.org


  Name: Group Europe
  Max Latency: 20.0 ms
  Min Latency: 0.0 ms
  Overall Group Cost: 2.2
  Group Size:  4
    planetlab-02.lip6.fr]
    planetlab2.dcs.ac.uk]
    planetlab-01.lip6.fr]
    planetlab2.uni-kl.de

  Total cost: 6.3
  Total query time:  0.095 s
```

(b)

Fig. 2. (a) Sample query in native SWORD syntax. The actual implementation of SWORD uses an XML representation of this information. (b) Sample SWORD response, indicating the IP addresses and hostnames of the resources found that meet the requested criteria.

shown in Figure 2(b). For clarity of presentation, we have converted the XML syntax that SWORD takes as input into a more human-readable format that structurally matches SWORD's actual XML syntax. Figure 3 shows the equivalent query using ClassAds syntax, minus the inter-node and inter-group constraints and rankings based on them.

A native SWORD XML query takes the form of an XML document with three sections. The first section of a SWORD query allows the user to request that the query be processed as a *continuous query* rather than as a default one-time-only query. The `RequeryInterval` directive specifies the frequency, in seconds, at which the query should be re-processed. When the user first issues their query, they receive in response the optimized mapping of available nodes to groups in the query, the nodes' resource measurements (this is optional and not shown in Figure 2(b)), and the penalty associated with the mapping. Every `RequeryInterval` seconds after the query was initially issued, SWORD recomputes the new optimal mapping and returns to the user the new mapping and the penalty difference between that mapping and the one originally returned. The user can monitor the stream of penalty differences returned by SWORD over time. If the difference reaches an application-specific criteria, perhaps exceeding a predefined threshold or exceeding a predefined threshold consistently for some period of time, the user can compare the original and new mappings to decide, for example, which application instances to migrate to new nodes. In the future we plan to explore the task of optimizing the new mapping for minimal application disruption; in other words, we will de-

```
{ [ name = ‘‘NA’’;
    nummachines = 4;
    constraint = load <= 2.0 && freedisk >= 500 && os == ‘‘linux’’ &&
       netdist(‘‘NorthAmerica’’) < 50;
    rank = (100.0 * (load-1 > 0 ? load-1 : 0)) +
           (100.0 * (1000-freedisk > 0 ? 1000-freedisk : 0)); ];
  [ name = ‘‘Europe’’;
    nummachines = 4;
    constraint = load <= 2.0 && freedisk >= 300 && os == ‘‘linux’’ &&
       netdist(‘‘Europe’’) < 50;
    rank = (100.0 * (load-1 > 0 ? load-1 : 0)) +
           (100.0 * (1000-freedisk > 0 ? 1000-freedisk : 0)); ];
};
```

Fig. 3. Sample query using ClassAds syntax.

termine the optimal mapping that migrates the fewest application instances while still producing a low-penalty configuration.

Recall that one of our design goals was to allow users to define topologies of interconnected groups. The second section of the SWORD query defines the groups. It specifies the number of nodes in each group, as well as the constraints on the per-node and inter-node attributes within each group. The value of an attribute can be a static value, *e.g.*, an operating system, an instantaneous measurement, *e.g.*, the node's current free memory measurement, or a statistical property of a base attribute, *e.g.*, the variance of a node's load over the past hour. SWORD supports floating point, string, boolean, and network coordinate datatypes. Network coordinate constraints allow the user to specify that a node must be located within a desired latency of any other node in SWORD. This feature is useful for deploying instances of a service near known user populations, as in our search engine and CDN examples, or near data sources, as in our Grid example. SWORD tracks the network coordinates of a set of reference nodes in common geographical regions, and uses the network coordinate distance between the candidate node and the reference node to determine whether the candidate node is within the desired latency of the specified region. Note that only the user-specified attributes matter in a SWORD query. If an attribute is not specifically defined in a query, it is not evaluated or even considered by the query processor.

The third section of a SWORD query specifies pairwise constraints between individual members of *different* groups. These constraints are used to ensure that at least one pair of nodes that exist in different (and disjoint) groups meet some desired requirement. For example, suppose a user requests two groups of nodes that are physically separated into two distinct geographic regions. An inter-group constraint defined in the third section of the query is then used to request that at least one network link between the groups has some minimum amount of available bandwidth. We note that because an arbitrary number of inter-group constraints may be specified, SWORD is sufficiently flexible to describe any overlay network graph—in the worst case, every node is its own group, and every edge is represented by a pairwise constraint between the two nodes that the edge connects.

To further clarify the structure of a SWORD query, consider our earlier example query shown in Figure 2(a) that shows a request for two groups of resources. The first group consists of four nodes in the North America (NA) group. The query requires that the operating system on these nodes be Linux, and that the node be located within 50 ms, calculated using network coordinates, of a reference SWORD

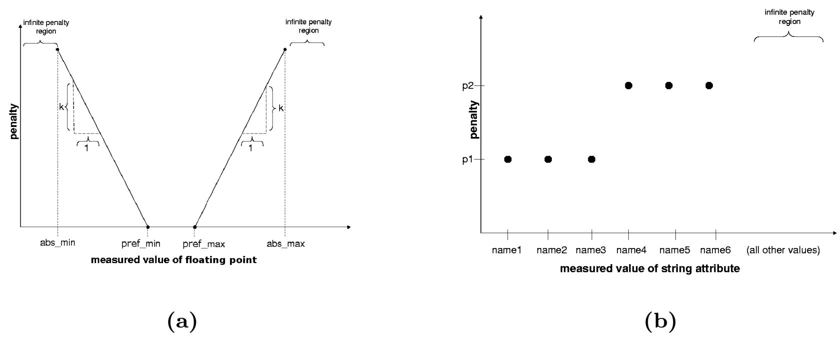


Fig. 4. (a) Floating point attribute penalty function. (b) String attribute penalty function.

node in North America. The query further requires that nodes' load be less than 2.0, free disk space be at least 500 MB, inter-node latency no more than 20ms, and inter-node bandwidth no less than 0.5 Mb/s. The Europe group is defined identically except that free disk space must be at least 300 MB. The "Preferred" clauses describe penalty functions, which we explain shortly. Also in our sample query, the third section defining inter-group constraints indicates that the user wants a pair of nodes such that one node is in the NA group, one node is in the Europe group, and the bandwidth between the two nodes is at least 3 Mb/s. Figure 2(b) shows a typical SWORD response to this sample query.

2.5 Penalty functions

Although SWORD is commonly used to simply find nodes meeting user specified requirements, a key feature of SWORD is its ability to rank node configurations according to user preferences. These preferences are defined by a "penalty function" associated with one or more attributes on a per-group basis. To simplify user query syntax, we allow penalty functions of a restricted shape that we believe is useful for approximately characterizing applications. In summary, the penalty function is defined by the "Required" and "Preferred" clauses for an attribute. The resulting configurations that SWORD returns match all requirements and are ranked based on their overall penalty or cost. The overall penalty associated with a node is taken as the sum of the per-attribute penalties associated with that node. The overall penalty of a group is computed by summing the per-node penalties of all members, in addition to the inter-node penalties of each link. The overall penalty of a configuration is the sum of the overall group penalty and the inter-group attributes across all groups. Note that for queries specified using the ClassAds syntax, the penalty function is defined by the "rank" statement, which defines the rank of a candidate node as a function of the per-node attributes of the node.

For *floating point* attributes, Figure 4(a) shows that the SWORD 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, a preferred region of zero penalty, and two constant-slope regions starting at the endpoints of the preferred region. Notice that the preferred region defines a sub-range of the required region. The penalty function for a floating point attribute takes the form

```
Required attr [abs_min, abs_max]
Preferred attr [pref_min, pref_max], penalty k
```

The k value maps a deviation from the preferred region to an abstract penalty unit. It therefore indicates how sensitive the application is to changes in the attribute's value—larger k denotes more incremental penalty per unit change in the attribute, while smaller k denotes less incremental penalty per unit change in the attribute—and defines the relative weighting of attributes.

Although most resource-related attributes are floating point quantities, users also have to ability to specify requirements and preferences for *string* attributes such as operating system and machine architecture. Figure 4(b) illustrates the penalty function for a string, which takes the form

```
Required attr [name1, name2, name3, name4, name5, name6]
Preferred attr [name1, name2, name3], penalty p1
Preferred attr [name4, name5, name6], penalty p2
```

This penalty function associates penalty $p1$ with discrete values $name1$, $name2$, and $name3$, and penalty $p2$ with discrete values $name4$, $name5$, and $name6$. Any values that meet the "Required" clause for an attribute but are not mentioned in a "Preferred" clause for that attribute are implicitly assigned a penalty of 0.

Boolean attribute penalty functions are of the same form as string attribute penalty functions, but the only allowed strings are "true" and "false." *Network coordinate* attribute penalty functions are of the same form as string attribute penalty functions, but in place of an arbitrary string they use a name that maps to a specific node, and a range of latencies from that node.

One extension to this work is the characterization of sensitivity to resource constraints for real applications. We believe that our assumptions that overall penalty is the sum of independent per-attribute penalties and that the penalty associated with an attribute can be approximated by a piecewise function are reasonable first approximations that are suitable for the majority of our users. While a simple piecewise function restricts the form of the penalty function for a small subset of our more advanced users, we currently believe that keeping the query syntax somewhat simple and intuitive for all users is important.

3. DESIGN AND IMPLEMENTATION

In addition to defining a resource specification language, Section 2 established the need for a logical database and query processor for storing and querying monitoring data, and an optimizer that finds an optimal mapping of resources to groups in the user's query. Given the design goals of Section 1, all of the components of SWORD must scale to large numbers of resources without sacrificing availability. For the logical database, this means we require sufficient storage (disk and memory) and network capacity to transmit and store the measurement updates from all nodes in the system. For the query processor, each node needs sufficient network capacity and memory to receive queries, filter nodes based on per-node attributes, and forward the results as appropriate. The optimizer requires enough CPU cycles and memory to complete at least a partial exponential search over the candidate nodes returned from the query processor. For per-node attributes, the storage, network, and memory requirements increases linearly as the number of nodes and metrics in the system increases. While per-node attributes scale linearly, the storage, network, and memory requirements for inter-node attributes scale exponentially as the number of nodes increase. In the following paragraphs, we describe how we achieve the desired scalability and availability in the implementation of our logical database and query processor in Section 3.1, as well as the design of our optimizer in Section 3.2.

3.1 Logical Database and Query Processor

SWORD's query processor is responsible for retrieving the per-node and inter-node measurements from the logical database needed to satisfy a query of the form described in Section 2.4. Recognizing that it may be useful for the query processor to treat per-node attributes differently from inter-node attributes, we split the retrieval of per-node attributes and inter-node attributes into two separate phases. In this section, we first describe several alternatives for storing and retrieving per-node attributes. Here the core algorithm of interest is *multi-attribute range search* to identify the set of nodes whose measurements match all ranges of per-node requirements for some group in the query, and that could therefore potentially be placed into one of the groups in the query response. In Section 3.1.3, we discuss alternatives for storing and retrieving inter-node attributes.

3.1.1 *Per-node Attribute Alternatives.* As previously mentioned, the core algorithm of interest for retrieving per-node attribute data is the multi-attribute range search. To better understand what is meant by multi-attribute range search, consider a query that requests a group of nodes that have load average between 0.0 and 2.0, and free disk space greater than 100 MB. Logically, to satisfy this query a join operation must be performed over the sets of nodes whose individual attributes fall within the desired ranges. One approach to satisfying this query is a centralized architecture where a central database collects and stores all nodes' reports of load and free disk space, and maintains indexes on load and disk space to quickly find nodes that meet the required ranges. One potential problem with this alternative is its lack of scalability as the number of nodes increases, which may also reduce the overall system availability. Thus, we look to distributed architectures in addition to centralized approaches, and compare their performance. Figures 5 and 6 show four different database and query processor architectures.

Before discussing our specific design alternatives, we must first define some basic terminology. A node that wishes to offer its resources through SWORD joins the SWORD infrastructure and collects resource monitoring data locally. This *reporting node* periodically sends a *measurement report* to one or more SWORD *servers* according to the query processor architecture in use. A node need not be part of the SWORD infrastructure to submit queries. To issue a query, a *client* node sends a query to any node in the SWORD infrastructure. This *query node* that receives the request acts as a proxy into the SWORD infrastructure, potentially issuing one or more SWORD sub-queries to one or more remote SWORD servers, again depending on the query processor architecture in use. The query node collects the necessary information to satisfy the query and returns the resulting node list back to the client, along with the measurement values of the attributes in the query. In our implementations, the reporting node, server, client, and query node roles are implemented on the same set of nodes.

Within this framework, we explore alternatives for the design of our logical database and query processor, focusing on how per-node attribute data is organized among the servers, and how a query is satisfied. How the data is organized dictates where a reporting node sends measurement reports, where and whether query nodes send sub-queries, and how the query node collects results. Overall, the alternatives we explore fall into three categories: distributed, centralized, and a hybrid between distributed and centralized. We implement four solutions in total: two distributed approaches that organize servers into a DHT (distributed hash table) overlay, one centralized approach that uses clusters of *fixed* servers, and one hybrid option that combines the distributed and centralized approaches, storing the measurement reports on nodes in the DHT overlay and storing on fixed servers an *index* that maps DHT key ranges to DHT server IP addresses.

A centralized architecture for SWORD consists of a single SWORD server that collects measurement reports as `<node,attribute,value>` triples from reporting nodes. It builds a database, each of whose "rows" contain all of the information from a single reporting node including its name, and one column for each reported attribute. Additionally, the server maintains indices over one or more attributes. Each index maps ranges of an attribute to the rows that currently record values of that attribute in the corresponding range. Answering a query then involves using the index to retrieve the rows for the value range of interest for one of the attributes of interest, and then filtering out the rows that do not meet the desired values of the other attributes. The final set of rows contains the measurement reports from the nodes that meet all criteria in the query. Since a single, central server does not scale well and is a single point of failure, we chose to implement a variation of this design that uses clusters of fixed servers rather than a single server to improve scalability and fault tolerance. In this approach, we assign the reporting nodes to one (or more for added fault tolerance) of the servers in our fixed server cluster. Then each of N fixed servers becomes responsible for maintaining all the current attribute values for (at least) $1/N$ of the reporting nodes. Assuming there is sufficient bandwidth and CPU power for the fixed servers, this solution provides much better scalability than a single server approach, since each fixed server requires $1/N$ of the bandwidth and CPU power that a single server would need for comparable performance.

The data storage and query processing becomes more complex when employing a distributed architecture. One option that we explore for distributed data storage, which is common in traditional distributed databases, is *dynamic range partitioning*: partitioning the data space according to dynamic attributes and ranges of values for those attribute. For instance, a subset of SWORD servers would become responsible for handling updates for all machines whose load average is between 0.0 and 1.0, a second subset for all machines whose load average is between 1.0 and 2.0, a third subset for machines that have between 100 MB and 200 MB of disk space, and so on. Given our target operating conditions, choosing a dynamic attribute such as load works just as well as a static attribute as the basis for range partitioning. Since we expire data quickly in our database to avoid maintaining stale information, it is more important to pick an attribute with a wide distribution of values for better load

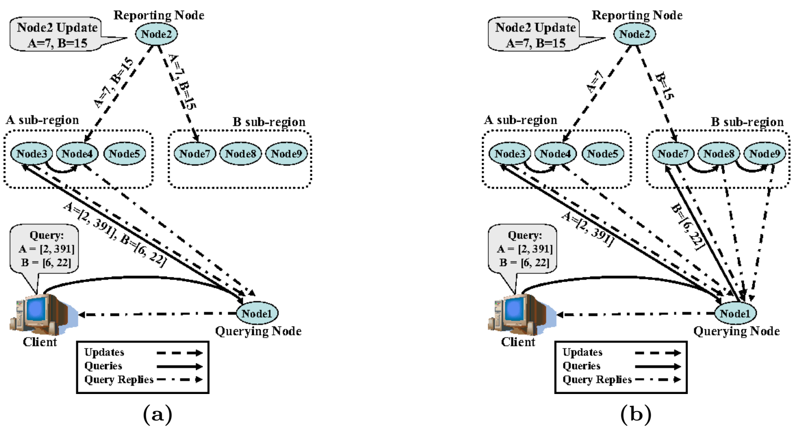


Fig. 5. (a) SingleQuery. For updates, the full report is routed to the node responsible for $a = 7$ and the node responsible for $b = 15$. For queries, the query is routed to a node chosen randomly among those responsible for values of a between 2 and 391, and forwarded to all other nodes responsible for those values. (b) MultiQuery. For updates, the a measurement is sent to the node responsible for $a = 7$, and the b measurement is sent to the node responsible for $b = 15$. For queries, the a portion of the query is sent to one randomly chosen node responsible for values of a between 2 and 391, and forwarded to all other nodes responsible for those values. The same is done simultaneously for the b portion of the query.

balancing among servers, than to pick an attribute that does not change frequently.

In the distributed approaches, we focus primarily on DHT-based range search algorithms because of their scalability, self-configuration, and high availability. These traits are a good match to our target of federated platforms that eschew centralized management, such as PlanetLab. Additionally, DHTs are well-suited to range partitioning, because they automatically partition a large (160 bit in our case) keyspace among servers. Our DHT-based algorithms are built on top of the Bamboo DHT [Rhea et al. 2004], though they could be built on top of any structured peer-to-peer overlay network. We leverage the DHT's ability to partition responsibility for the keyspace among servers, to deliver to the responsible node a message addressed to a key ("key-based routing" [Dabek et al. 2003]), and to enable, through "successor pointers" that organize the nodes into a linked list sorted by ascending key ranges, the visiting of the nodes responsible for a contiguous range of keys. On top of the DHT's key-based routing interface we build our own soft-state distributed data repository. All non-DHT messages are sent using UdpCC [Rhea et al. 2004].

The principal alternatives we consider for the distributed approaches also explore how the amount of information stored in a measurement report impacts performance. The information stored in a measurement report is somewhat dependent on what the resource monitors are measuring. We currently use a bootstrapping mechanism at startup to determine the list of available attributes being reported by the resource monitors. This allows us to inform the reporting nodes of the "useful" attributes for future measurement reports. As detailed below, one alternative for the measurement report is to include information only for the attribute of interest to the particular SWORD server. A second alternative is to include all current attribute values for the particular reporting node. These alternatives have implications for how queries are performed, and have varying levels of performance depending on the workload and deployment scenarios. The specific details of the design of our four approaches are as follows.

— Figure 5(a) illustrates **SingleQuery**. A reporting node sends n measurement reports, each containing the n attribute values it has measured, to n SWORD servers. Each report is routed to a DHT key computed, as described later in this section, from the value of one of the attributes in the update. While updates are large (size n), in this approach a multi-attribute query need only be sent to the set of servers responsible for the target range of one of the queried attributes. In particular, the query is first routed to any node responsible for any key in the query range of any attribute in the query, and it is forwarded along DHT successor pointers to all other nodes responsible for keys in the query range of that attribute. This technique is similar in principle to how Mercury [Bharambe et al. 2004] performs multi-attribute distributed range queries and how eSearch [Tang and Dwarkadas 2004] performs multi-keyword full text searches over a DHT.

— Figure 5(b) illustrates **MultiQuery**. The reporting node places a single value in each measurement report. Thus, a node reporting n attributes transmits n 1-attribute measurement reports to n SWORD servers, with the measurement report routed to the DHT key computed from the value of the attribute in the report. This approach has the potential to reduce update bandwidth consumption relative to SingleQuery. The downside is that to perform a query, one set of SWORD servers must be queried for each attribute in the request. The query node sends messages to the servers in parallel and intersects the returned nodes to find those that satisfy all attributes in the query. Because each server only stores information about one attribute, each server can only filter on one attribute, so this approach can potentially return many more nodes than SingleQuery, in which each server can filter on all attributes. This approach resembles in some ways how multi-attribute searches are performed by XenoSearch [Spence and Harris 2003] and how keyword search is performed in [Reynolds and Vahdat 2003].

— Figure 6(a) illustrates **Fixed**. This approach is basically a centralized data-center model with a varying number of servers. To send an update, a SWORD node sends one copy of its measurement report to one of n infrastructures servers that is assigned at random when the reporting node starts up, so that approximately m/n SWORD nodes are assigned to each server when the reporting node population is m. (Note that if redundancy is desired for increased fault tolerance, the reporting node sends its report to kn servers, and each server stores information for km/n nodes, where k is the level of redundancy.) A server acting as a query node forwards the query to the remaining $n-1$ servers, collects the results, and returns the results to the querying node or client.

— Figure 6(b) illustrates **Index**, which is a hybrid between the Fixed and SingleQuery approaches. The fixed infrastructure servers hold an index containing the mapping of contiguous DHT key ranges to the IP address of the DHT node responsible for that range. Each DHT node periodically informs one of the index servers of the range of keys for which it is responsible. The keyspace index is range-partitioned among the fixed servers. Updates are handled as with the SingleQuery and MultiQuery approaches. A query is sent first to the index server(s) responsible for the key range of interest, and those index servers then forward the query di-

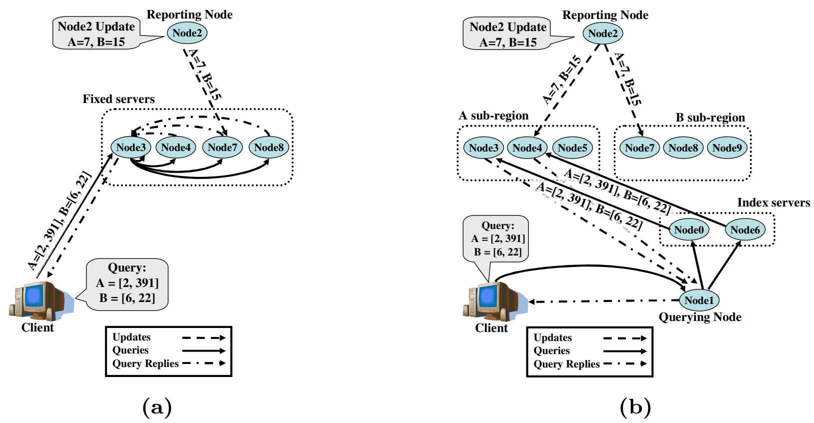


Fig. 6. (a) Fixed. For updates, the full report is sent to one of N servers chosen at random. For queries, the client sends the query to one of N servers chosen at random. The server that receives the query, called the "querying node", sends the query to each of the other $N-1$ servers. (b) Index. Updates are handled identically to SingleQuery. A query is first routed to the set of index servers responsible for the range specified in the query, and the index servers forward the query to the appropriate nodes in the DHT.

rectly to the DHT nodes responsible for the requested key range(s) without having to route through the DHT.

For all of the DHT-based approaches, one implementation detail relates to how the querying node decides it has received all responses. For the Fixed architecture, the query is complete when a response has been received from all queried servers. For the other architectures, the querying node does not know ahead of time to which, or how many, servers the query will be routed. But the querying node does know the lowest and highest DHT keys corresponding to the range of values to be queried. Additionally, all nodes send back the range of DHT keys for which they are responsible when they send a reply to the querying node. An empty response is returned if the DHT node has no matching nodes. Thus, the querying node can keep track of gaps in the query's key range from which no responding node has yet indicated responsibility. As responses are returned, the missing ranges of the query key range will shrink until a response has been received from the nodes responsible for each piece of the query key range. At that point the response set is complete. The query may also eventually time out if some messages are lost, at which time the query processor will return the partial results received thus far.

Nodes send their measurement reports at a frequency interval of their choosing. The interval chosen is indicated as a parameter (called the TTL) in the measurement report. When a report arrives at a server, the server records the update in memory—thus reports are soft state. The servers time out these reports when the TTLs expire so that when a reporting node fails, that node will eventually no longer appear in the result set of any query. Our use of soft state also provides a low-overhead mechanism for recovery from failures within the server infrastructure in the DHT-based approaches: in our DHT-based approaches, 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 the report. This soft-state-with-timeout mechanism requires that the reporting nodes re-publish their measurement reports more often than their chosen timeout interval. However, since node measurements are likely to change over fairly short timescales, frequent re-publishing is arguably necessary for accurate query answering anyway. The alternative, using the reliable DHT storage layer rather than our memory-only storage would add unnecessary network, processor, and memory overhead by replicating each piece of data on multiple nodes, only to change that data soon thereafter.

3.1.2 *DHT Keys and Load Balancing.* An important design issue when applying DHT-based range search techniques to resource discovery is how to construct the 160-bit DHT key corresponding to a measurement value. Recall that it is the SWORD node that "owns" this DHT key that will receive the corresponding measurement update. To accomplish this mapping, we associate with each per-node attribute A a monotonically non-decreasing function $f_A(x)$ that maps a value from the range of A to a DHT key. The f_A function can be user-defined, or the user can choose to use a per-datatype default function built into SWORD.

A reporting node sends measurement reports to servers by routing each report to a DHT key $f_A(x)$ for each attribute A in the report and its corresponding value x. A querying node sends a query to one (in SingleQuery) or multiple (in MultiQuery) servers. Taking the example of SingleQuery, the querying node chooses one attribute B from the query as the "range search attribute." It computes $f_B(m)$ and $f_B(n)$ where m is the lowest value in the queried range for attribute B and n is the highest value in the queried range for attribute B. The query node routes the query to a randomly chosen key within the range $f_B(m)$ to $f_B(n)$. The receiving node replies to the query using the information it stores and forwards the query to its successors, which, by the structure of the DHT, form a linked list of Node IDs in sorted order. The search range wraps around to the low end of the range upon reaching the node responsible for $f_B(n)$. This process continues until all nodes responsible for the range of keys $f_B(m)$ to $f_B(n)$ are visited and have returned to the querying node the matching results they store. Because our f functions are monotonically non-decreasing, they are guaranteed to map contiguous attribute values to a contiguous range in the DHT, making it feasible to follow successor pointers in the DHT to cover all nodes with attributes in a user-specified range.

Our f mapping constructs the key from three parts: high-order attribute bits, middle value bits, and low-order random bits. The first bits of the DHT key are "attribute bits." Each attribute is mapped to a setting of the attribute bits which does not need to be unique. These attribute bits partition the keyspace into "sub-regions," each responsible for one or more attributes. Sub-regions allow us to limit the worst-case number of nodes that might be visited to answer a distributed range query—an unconstrained search on a single attribute will visit at most all the nodes in the sub-region for that attribute—thereby helping to bound query latency. Although all nodes must agree on the number of key bits used as attribute bits, the mapping of attributes to attribute bits can be computed autonomously, for example, by hashing the name of the attribute.

The remaining bits of the DHT key are divided between "value bits" and "random bits" depending on the number of values each attribute commonly takes on. The value bits represent the value itself, while the random bits spread instances of the same value among multiple nodes for load balancing of updates. For example, a boolean-valued attribute would be represented using one value bit followed by $160 - a - 1$ random bits, where 160 is the number of bits in a key and a is the number of attribute bits. This randomization would serve to spread "false" values over half of the nodes in that attribute's sub-region, and "true" values over the other half of the nodes in that attribute's sub-region. When a node issues a query, the random bits are set to all 0 when computing the key for the lowest value in the range, and are set to all 1 when computing the key for the highest value in the range. In this way, the query will search all keys that could possibly have been generated from the desired range.

Our current implementation uniformly maps the expected range of a given *floating point* attribute to the number of bits allocated to value bits. For example, if 2 bits are used for value bits and the value represented is an integer percentage (0-100), then the mapping function might map $0 - 25$ to `00`, $26 - 50$ to `01`, $51 - 75$ to `10`, and $76 - 100$ to `11`. If the distribution of update values is known ahead of time, then more frequently-occurring values can be "spread" over more keys relative to less frequently-occurring values, to improve load balance for updates. Extending the previous example, if $0 - 25$ occurs three times more often than $26 - 100$, then the mapping function might map $0 - 8$ to `00`, $9 - 16$ to `01`, $17 - 25$ to `10` and $26 - 100$ to `11`. Because we only require that f_A be monotonically nondecreasing, all values less than the lowest value in the expected range can be mapped to all 0 value bits, and all values greater than the largest value in the expected range can be mapped to all 1 bits. The expected value range does not need to be known ahead of time to achieve a good spreading of input values; only the range of "typical" values needs to be known, and outliers are mapped to the two extreme endpoints.

The default function for *booleans* is as described earlier. The default function for *strings* forms the value bits from the high-order bits of the ASCII value of the string, allowing string prefix searches, such as "all IP addresses that begin with 128.112." Finally, the default function for *network coordinates* generates the value bits using the z-coordinate [Jagadish 1990] linearization of the 3-dimensional network coordinate of the client node. (The z-coordinates are determined using an out-of-band mechanism that maps geographic names to corresponding z-coordinates.) This transformation enables multidimensional range search in the n-dimensional network coordinate space via a linear search over the z-coordinate attribute. This mapping is especially useful for the MultiQuery approach because it allows one range query to be issued, over the synthetic z-coordinate, instead of three, namely one over each of 3 dimensions.

In summary, attribute bits reduce conflicts between identical values of different attributes, random bits reduce conflicts between identical values of the same attribute, and value bits can be used to spread potentially non-uniform value ranges to more uniform DHT key ranges. These load balancing techniques are *passive* since they do not require nodes to measure or react to their load. A second, *active* layer of load balancing [Karger and Ruhl 2004; Bharambe et al. 2004] can further enhance load balance by observing load distributions on nodes and remapping Node IDs accordingly. The active and passive load balancing techniques are orthogonal.

3.1.3 *Inter-node Attribute Alternatives.* The process we have described thus far retrieves the identities of all reporting nodes matching all *per-node* requirements in the query, along with the values of those attributes. A number of alternatives also exist for handling *inter-node* attributes, which are attributes defined between pairs of nodes, for example inter-node latency or available bandwidth. These alternatives revolve around which nodes measure and report inter-node attributes, and how inter-node measurements are stored and queried. With respect to the first issue, we can have all nodes measure and report inter-node attributes, or we can elect a subset of nodes as "representatives" for other nodes that are likely to have similar values for the inter-node attribute in question. With respect to the second issue, we can store inter-node measurements either within the DHT or externally.

The key distinction between per-node attributes and inter-node attributes (and the main reason we separate the discussion into two different sections) is the amount of data that must be stored as the number of resources in the system increases. While per-node attributes scale linearly with the number of nodes in the system, inter-node attributes scale exponentially. To reduce the resource consumption of gathering $O(N^2)$ inter-node measurements, SWORD leverages the observation that nodes typically fall into equivalence classes for several inter-node attributes. For example, the latency between Node A in Autonomous System 1 (AS1) and Node B in AS2 is often approximately equal to the latency between any node in AS1 and any node in AS2. (While there are certainly exceptions to this generalization, on target infrastructures such as PlanetLab, we find that this is true in most cases.) SWORD therefore allows arbitrary groups of nodes to define a "representative" node that collects inter-node measurements on their behalf. Choosing appropriate representatives is an orthogonal issue that we do not address and might leverage existing work on network-aware clustering [Krishnamurthy and Wang 2000; Chen et al. 2004]. The mapping from each node to its representative is one of the per-node attributes that nodes report; this is essentially an "object location" pointer. We allow arbitrary equivalence classes to be defined, perhaps on granularities smaller than ASes.

The distributed query that retrieves inter-node measurements could take a number of forms. For example, we could use an approach similar to our per-node attribute range search but use as the DHT key an inter-node measurement range, for example "10-20 ms", and use as the values stored at that key the list of node pairs whose inter-node measurements are within that range, for example all node-pairs whose inter-node latency is between 10 ms to 20 ms. The query would visit the DHT keys corresponding to inter-node value ranges specified by the inter-node requirements of the query, and the servers responsible for those keys would return all nodes such that both endpoints are nodes returned from the per-node attribute query step. Note that choosing the "resolution" of these groups well requires some knowledge of the distribution of inter-node measurements.

Alternatively, the DHT key could be a mixture of an IP address and an inter-node measurement range, for example "10.0.0.1, 10-20 ms". We could then use as the values stored for that key the list of nodes whose inter-node measurement from the indicated node is within that range, for example all nodes that are within 10 ms to 20 ms from the node 10.0.0.1. In this case, the query would visit the DHT keys corresponding to inter-node value ranges specified by the inter-node requirements of the query for the nodes that were returned in the per-node attribute query step.

A third approach would be to use as the DHT key simply the IP address I of a representative, and use as the values stored for that key a list of all other representatives and their inter-node measurement from the representative with IP address I. This approach does not allow us to perform a distributed *range* query, only a distributed query, with the range filtering done on the querying node once it has received all pairwise measurements among all representatives of interest.

Our distributed query processor implementation actually uses a variant of the third approach: instead of storing inter-node measurements in the DHT, representatives themselves store the inter-node measurements that they collect. This technique saves them the bandwidth of having to publish a potentially large number of inter-node measurements into the DHT. Such savings are particularly beneficial if representatives measure inter-node attributes often, for high accuracy. Thus, in SWORD, after a querying node receives the node reports R from the per-node attribute range query, the querying node issues a second distributed query, to the representative nodes indicated in R, to obtain the inter-node attribute(s) of interest among those representative nodes. This two-step process is essentially a "join" operation. To bootstrap this process, each node in the system need only know the identity of its own representative. Each node periodically reports the identity of its representative as part of its measurement reports. When a representative node boots, it performs a standard SWORD distributed query to find the identities of all other representatives in the system, and begins measuring to them.

3.2 Optimizer Implementation

Section 3.1.1 describes how the query processor obtains and uses the per-node measurements to create a set of candidate nodes, and Section 3.1.3 discusses how to gather the corresponding inter-node measurements. It is important to realize that the query processor designs previously discussed do not filter based on inter-node measurements. They also do not compute any penalties. They simply perform a disitributed range search for per-node attributes, and then gather the inter-node measurements for the candidate nodes that meet the per-node constraints specified in the query. These candidate nodes and their corresponding per-node and inter-node measurements are then fed to the optimizer. It is the optimizer's role to determine an appropriate mapping of nodes to the groups specified in the query. The optimizer computation is not itself distributed, as it runs only on the "query node" that receives the initial query from the client. However, because a client may contact any SWORD node as its proxy, the computation is effectively distributed on a per-request basis, assuming nodes choose proxies randomly.

The optimizer proceeds in several steps. The input to each step is a list of nodes organized into groups, and the output is also a list of nodes organized into groups. Some phases use heuristics to improve their running time at the expense of solution optimality; we describe those heuristics below, and analyze their impact on running time and solution optimality in Section 4.2.

Phase 1 computes per-node penalties based on the penalty functions defined in the user's query. We take as input a list of candidate nodes from the query processor, and the corresponding per-node and inter-node measurements. The optimizer creates, for each group in the user's query, a list of candidate nodes sorted by the per-node penalty associated with placing that node in the group. Note that the per-node penalty associated with placing a node in a group is not affected by the choice of other nodes in the group. In other words, we assume that the per-node penalties are independent from each other.

Phase 2 computes per-group penalties using the per-node penalties from Phase 1, in addition to the inter-node penalties associated with each group that are computed in this phase. For each group in the user's query, we create combinations of nodes from the Phase 1 candidate list that also meet the inter-node constraints specified in they query. These combinations of nodes are called "candidate groups." Once all possible candidate groups have been formed for each group specified in the user's query, the optimizer sorts the candidate groups based on total group penalty, which is computed by summing the per-node attribute penalties across all nodes in the group plus the inter-node attribute penalties. Note that given a list of n candidate nodes for a group that requires k nodes, there are $\binom{n}{k}$ possible candidate groups that must be considered, though all candidate groups do not necessarily satisfy the inter-node requirements.

Creating groups of a specific size that satisfy the inter-node constraints is an NP Hard problem; it is an instance of the *k-clique* search problem. Thus, as an optimization, we form groups that use the lower-penalty nodes in the list passed in from Phase 1 before we use the higher-penalty nodes in the list. We can do this easily since Phase 1 sorts the nodes based on per-node penalties. Furthermore, we allow the user to specify a maximum per-group running time for this phase of the computation. Therefore, if the computation is terminated for a particular group before all candidate groups are enumerated, the candidate groups that have been formed thus far will have low per-node penalties, and thus are likely to yield groups with low overall (per-node plus inter-node) penalty. This heuristic will not help, however, if the inter-node constraints are given much more weight in the user's query than per-node constraints. In that case, groups with low per-node penalty may not have low overall penalty, and the candidate groups with low overall penalty will never be formed if the computation is cut short in the way we described. The output of Phase 2 is one list of candidate groups per group in the user's query. Each list is sorted by overall group penalty before being passed to Phase 3.

Phase 3 computes total solution penalty based on the already-computed per-node and inter-node (also called per-group) penalties, plus inter-group penalties computed in this phase. We enumerate a list of "candidate answers" to the user's query. These are combinations of candidate groups from Phase 2 that satisfy the query's inter-group constraints; they are therefore potential "solutions" to the user's query. For example, given lists from Phase 2 of size a, b, and c, Phase 3 could generate up to $a * b * c$ candidate answers, though not all may satisfy the inter-group requirements.

Creating candidate answers that satisfy the inter-group constraints is again an NP hard problem. Thus, as an optimization, we form solutions that use the lower-penalty candidate groups from each list passed in from Phase 2 before we use the higher-penalty groups from those lists. As in Phase 2, we impose an optional user-specified limit on the running time of this phase. If the computation times out before the full search is complete, the candidate answers that have been formed up to that point will have low per-group penalties, and thus are likely to yield answers with low overall (per-group plus inter-group) penalty. Analogous to our caveat on this optimization in phase 2, this heuristic is not useful if the user's query weights inter-group constraints much more heavily than per-group constraints. In that event, groups with low per-group penalty may contribute significant penalty to the overall solution, and the candidate answers with low overall penalty will never be formed if the computation is cut short. The output of Phase 3 is one or more "candidate answers" along with their associated penalties; these solutions are returned to the user.

In Section 4.2, we examine several heuristics for reducing the optimizer's running time, possibly at the expense of solution optimality. **Three second timeout** simply runs the exponential search for three seconds and returns the best solution found at the end of that time. The three second time budget is evenly divided between Phase 2 and Phase 3 described above. **Top half of candidates** searches only the lowest-penalty half of the candidate groups. **Top 5 candidates** tries to shortcut the previous heuristic by eliminating all but the top 5 candidates for each group before running the search. Finally, since the groups are sorted based on overall penalty, and the lowest penalty groups are processed first, the first answer found may be relatively good compared to the optimal solution. Thus the **First answer** heuristic runs the exponential search but stops as soon as the first valid solution is found.

3.3 PlanetLab Implementation and Deployment

SWORD has been running as a publicly-accessible service on PlanetLab since June 2004. A SWORD instance runs on every PlanetLab node, periodically issuing measurement reports. A user creates a query by either manually writing XML or using the SWORD web interface which automatically generates XML. The web interface (or publicly available command line client) then makes a TCP connection to any SWORD instance, and sends the text of the query over the connection. The contacted SWORD instance invokes its query processor which issues a search for nodes meeting the per-node criteria and gathers the corresponding inter-node requirements, and invokes its optimizer (locally) using the result of the search. That node then returns to the user over the same TCP connection the result, which is a list of nodes, the groups to which they have been assigned by the optimizer, total penalties accrued, and if desired, the raw node measurements that were used in making the assignment.

SWORD can also be used programmatically. For example, the Plush [Albrecht et al. 2006] distributed application management system takes a user defined description of a distributed application, and then automates the process of finding resources, preparing them for execution, running any executables, and recovering from failures for the duration of an application's lifecycle. For PlanetLab users, Plush has the ability to use SWORD as its resource discovery and service placement component. Thus Plush "closes the loop" of the application deployment process, by automatically instantiating a user's application on the nodes returned by SWORD. The Bellagio [AuYoung et al. 2004] microeconomic-based resource allocation system similarly uses SWORD as its resource discovery component. Motivated from the development and use of these technologies, we present quantitative evidence that applications benefit from using resource discovery services in making deployment decisions in [Oppenheimer et al. 2006].

SWORD currently uses four data sources for resource measurements on PlanetLab: a Vivaldi [Dabek et al. 2004] network coordinates implementation that is built into SWORD, the Ganglia [Massie et al. 2004] daemon on each node that collects node-level measurements, and the "CoTop" tool [Pai] invoked on each node to collect slice-level measurements from the slicestat [Chun] sensor. Because the number of nodes on PlanetLab is relatively small, we are able to configure every node as a "representative" without imposing extraordinary inter-node measurement or storage overhead.

For SWORD's PlanetLab deployment, we extended the query language to allow users to specify, for each group in their query, the maximum number of nodes that can be assigned to that group from any single PlanetLab site. This feature allows the user some protection from correlated failures that disconnect all of a site's nodes from the network, such as a power failure or failure of the site's Internet gateway. For example, a user might request 10 nodes from 10 different sites instead of running the risk that all 10 nodes will come from the same site. A "site" can be defined arbitrarily; we use the simple approximation of assigning a site ID to each node as the hash of its DNS suffix, but other approximations such as the node's AS number could be used, or a delegation model could be used in which each site that joins PlanetLab is assigned a site number by a central authority, and the site then autonomously creates sub-sites by appending additional digits to their site number.

To add an attribute to SWORD, a reporting node's administrator may choose to use one of the four default f_A functions or a custom-written function. Our current implementation trusts administrators to supply well-behaved f_A functions when they add attributes with custom f_A functions to the system; protecting against malicious or misbehaving user code that might infinite loop or crash is left as future work. Once a new attribute is installed on a reporting node, SWORD itself can be used to distribute the attribute's identity and its f_A function: reporting nodes can include a list of the names of the attributes they report, and the corresponding f_A's, in their measurement reports. All nodes can periodically probe the full range of this attribute to retrieve all other nodes' reported attributes and f_A's. If one node is the "seed" node that introduces an attribute, all nodes will discover the new attribute and its f_A function the next time they conduct such a probe query.

4. PERFORMANCE EVALUATION

We are interested in evaluating the following properties of SWORD: (i) How does the performance of the various DHT-based range search approaches compare to one another and to the fixed-servers implementation? (ii) How does workload intensity affect performance? (iii) How do our optimizer heuristics impact optimizer performance and accuracy compared to performing the full exponential search? (iv) How well does our system perform on PlanetLab?

In Section 4.1 we focus on the first two questions, describing the results of an emulation-based evaluation of our distributed query processor configurations. In Section 4.2 we address the third question, examining the performance of the optimizer. Finally, in Section 4.3 we address the question of performance on PlanetLab.

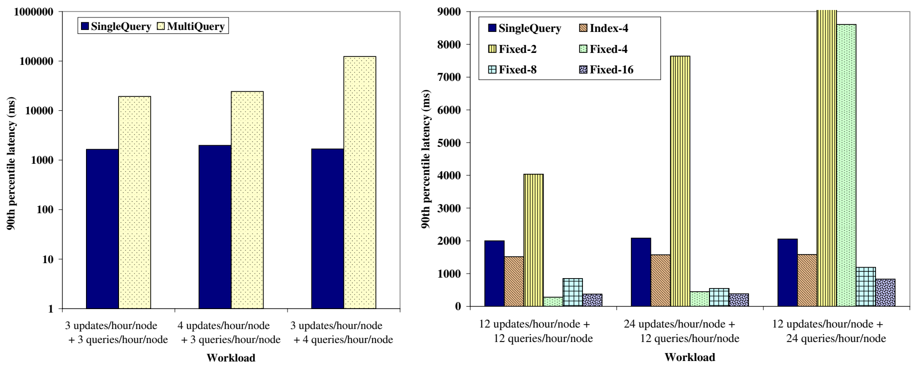
4.1 Emulation-based Evaluation

Our emulation experiments focus on the performance of SWORD's distributed query processor that retrieve per-node and inter-node measurements. We choose query latency as our performance metric because SWORD users may periodically re-query SWORD to adapt their application to changing node and network conditions, or as the resource needs of their application changes. For our emulation experiments, we run the distributed query processor to collect per-node and inter-node measurements, but we do not invoke the optimizer on the result. We evaluate optimizer performance separately in Section 4.2, and the end-to-end performance of the entire system, on PlanetLab, in Section 4.3.

We evaluated SWORD's query processor on a cluster of 40 IBM xSeries PCs with Dual 1 GHz Pentium III processors, 1.5 GB of RAM, and Gigabit Ethernet. We used ModelNet [Vahdat et al. 2002] with an INET topology [Chang et al. 2002] consisting of 10,000 transit nodes, 1,000 (virtual) client nodes, and 8 client nodes per stub. In ModelNet, packet transmissions are routed through *emulators* responsible for accurately emulating the hop-by-hop delay, bandwidth, and congestion of a network topology. Propagation delays in the network topology are calculated based on the relative placement of the network nodes in the plane by INET. Groups of network nodes are classified as being client, stub, or transit depending on their location in the network. Transit-transit links were given 150 Mb/s (OC3) bandwidth and client-stub links 384 Kb/s bandwidth. Latencies were based on the INET topology. 1/32 of the nodes were chosen as "representatives." When evaluating the Fixed and Index approaches, the infrastructure servers were grouped into 4-node stub domains each with 150 Mb/s, 1 ms latency connections to their upstream transit node. For the DHT-based approaches, we used the Bamboo DHT.

Our baseline workload consisted of measurement updates and queries issued by each of 1000 virtual nodes. The content of updates were taken from a representative one-hour portion of a trace of Ganglia measurements collected from all live PlanetLab nodes every 5 minutes between July 2004 and October 2004. Updates contain 32 metrics collected by Ganglia during that time period, along with Vivaldi network coordinates and several debugging attributes, for a total of 40 attributes.

Queries contained five per-node attributes—fifteen-minute load average, free disk space, free memory, network receive bandwidth, and network transmit bandwidth—and one inter-node attribute, inter-node latency. Queries were formulated according to a distribution such that they requested a minimum amount of disk space that is Zipf distributed between 10 MB and 100 MB (biased toward the high end of the range); a fifteen-minute load average less than a uniformly distributed value between 0 and 5.0; a minimum amount of free memory that is Zipf distributed between 0 MB and 48 MB (biased toward the high end of the range), bytes per second in and out (competing traffic) that is no more than 0.1 MB/s for half of the queries and unconstrained for the other queries, and inter-node latency between 0 ms and 1000 ms. Because our trace contained valid data for only 124 PlanetLab nodes, we emulated a 1000-node system by replicating each of the 124 entries an average of 8 times. The median number of nodes returned per query was 120 and the 90th percentile was 160. In the DHT approaches, attribute bits are assigned so



(a) **(b)**

Fig. 7. (a) 90th percentile latency of the SingleQuery vs. MultiQuery approaches. (b) 90th percentile latency of range query approaches.

that each of the 40 attributes is mapped to a sub-region containing 25 nodes.

We examined the four query processor architectures from Section 3.1 under a variety of workloads. For evaluating the Fixed and Index approaches, we modified the emulated topology so that the data servers and index servers, respectively, were each placed into $N/4$ groups of 4 nodes each, where N is the number of Fixed (2, 4, 8, or 16) or Index (4) servers used. (For Fixed-2, one group of 2 nodes was used.) As previously mentioned, each group was given a 150 Mb/s, 1 ms latency network connection to its upstream transit node, and servers within a group were given 150 Mb/s point-to-point communication links. This configuration is intended to emulate an environment in which a service provider has placed the N servers in $N/4$ geographically distributed, well-connected co-location centers. For Index, we chose to use 4 index servers because preliminary results showed that using fewer worsened performance, while using more did not improve performance significantly.

4.1.1 *Distributed Query Latency.* Figure 7(a) shows the impact of range-search approach and workload intensity on the latency to satisfy the range query for per-node attributes in both the SingleQuery and MultiQuery approaches, with a workload of 3 or 4 updates/hour/node and 3 or 4 queries/hour/node evenly generated by all nodes. At higher workload rates, our emulation cluster's processors became overloaded for the MultiQuery approach. The primary reason for the difference between SingleQuery and MultiQuery is that the network bandwidth consumed by the larger number of nodes returned to the querying node by MultiQuery compared to SingleQuery creates sufficient congestion to overwhelm the benefit MultiQuery derives from sending only one attribute per measurement report. We would expect that substantially increasing the number of attributes per update, substantially decreasing the number of attributes per query, or decreasing the selectivity of queries, would serve to reduce this performance difference somewhat. However, we believe our choice of these values is reasonable for an experimental resource discovery workload. Since these experiments reveal that SingleQuery clearly outperforms MultiQuery for our "typical" resource discovery workload by at least an order of magnitude, so we do not consider MultiQuery in the remainder of our evaluation.

Figure 7(b) shows the impact of range-search approach and workload intensity on the latency to satisfy the range query for per-node attributes for the SingleQuery, Index, and Fixed approaches, with a workload of 12 or 24 updates/hour/node and 12 or 24 queries/hour/node, evenly generated by all nodes (except the servers in the Index and Fixed approaches). We see that Index always outperforms SingleQuery. This result is reasonable because queries in Index take three hops in parallel—one to the index server(s), one to the DHT server(s) storing measurements, and one back to the querying node—while a query in SingleQuery visits up to 25 nodes.

The Fixed approaches vary greatly in performance. With two servers (Fixed-2), Fixed underperforms the DHT-based approaches (SingleQuery and Index) for all workloads. This is likely due to the limited bandwidth and processing power that using only 2 servers provides. With four servers (Fixed-4), Fixed outperforms the DHT-based approaches for all but the 12 updates/hour + 24 queries/hour workload. Again the main limitation for the more intense query workload was the lack of sufficient bandwidth and processing power on the server nodes. With eight or sixteen servers (Fixed-8 or Fixed-16), Fixed always outperforms the DHT-based approaches (SingleQuery and Index). This result suggests that a more centralized approach using a fixed infrastructure cluster with a relatively small number of nodes and high-bandwidth network connections better supports the resource discovery workload that we tested than an infrastructure based on end-nodes organized into a DHT. As predicted, the main problem experienced with the centralized architecture was lack of scalability due to insufficient network bandwidth, and the smaller cluster configurations that performed poorly did so because of network congestion, which we verified by examining router queues.

Although the above workloads included retrieval of inter-node measurements from representatives, our graphs show only the time for the range search over per-node measurements. We found that the end-to-end query processor latency—in other words, including per-node attribute and inter-node attribute retrieval—followed exactly the same trend as the per-node attribute latency alone. The explanation for this observation is that inter-node measurement retrieval is affected by the same congestion as per-node measurement retrieval. We found that the amount of time to retrieve inter-node measurements was about 600 ms for the DHT-based approaches, about 450 ms for the Fixed-8, Fixed-16, and Fixed-4 workloads that outperformed the DHT approaches in Figure 7, and was 1000 ms or greater for the remaining Fixed variants and workloads.

Our use of representatives has a significant impact on 90th percentile end-to-end query processor latency. For the 12 updates/hour/node, 12 queries/hour/node workload, we compared selecting all nodes in the system as representatives to selecting only half of them. This latter configuration reduced query latency by more than 70% for the Fixed-2 and Fixed-4 implementations, where network congestion at the fixed servers is high and bandwidth consumption savings are thus very valuable, and by 7% in the SingleQuery approach.

4.2 Optimizer Performance

Due to the complexity of the searches that the optimizer performs, it is beneficial to consider some heuristics that reduce the size of the space that must be searched. Although more complicated algorithms exist for solving this optimization problem,

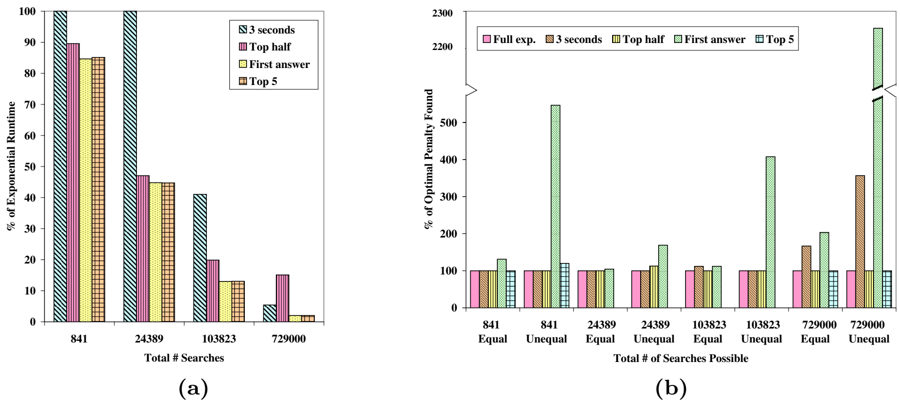
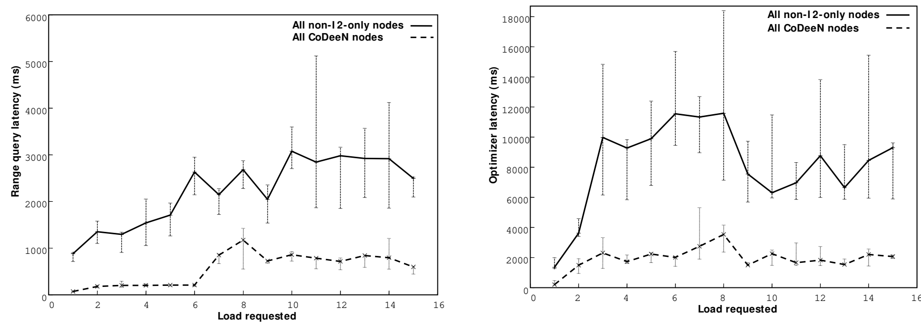


Fig. 8. (a) Runtime of optimizer using different heuristics shown as a percentage of the runtime to complete the full exponential search. The x-axis shows the total number of possible group combinations that would be checked if the complete exponential search were run. (b) Accuracy of optimizer heuristics relative to the optimal solution found in the full exponential search. The x-axis shows the total number of possible group combinations that would be checked if the complete exponential search were run. The "Equal" bars represent queries where the penalties assigned to all attributes are weighted equally. The "Unequal" bars represent queries in which the penalty assigned to the inter-group constraint is 10 times greater than the per-node and per-group inter-node attributes' penalties. A missing bar indicates that no solution was found.

we chose to evaluate some simple heuristics that attempt to shortcut the full exponential search while still finding a solution that is close to the optimal. The specific heuristics that we evaluated were described in Section 3.2.

We measured the performance of the optimizer on a single 3 GHz Pentium 4 processor node with 512 MB of RAM. The workload consisted of queries containing 2 or 3 groups. The queries required load less than a value between 0 and 5, inter-node latency within each group less than 100 ms, and at least one inter-group link between 0 ms and 200 ms. The "preferred" attribute values included load less than 1, inter-node latency within each group between 5 ms and 50 ms, and at least one inter-group link between 50 ms and 100 ms. The underlying data used for all queries was taken from a snapshot of Ganglia system resource statistics and all-pairs-ping measurement from PlanetLab in October 2004. The queries were satisfied by less than 20% of the possible group combinations for all experiments. We considered two settings for the penalties of the queries. For half of the experiments, all attributes (per-node load, pairwise intra-group latency, and inter-group latency) were assigned equal penalties. For the other half, the inter-group latency attribute was assigned a penalty 10 times greater than the per-node and inter-node attributes to explore the "pathological" case where the sorting based on per-node and per-group total penalties does not provide any benefit to the complete search.

To quantify the benefits of the various heuristics, Figure 8(a) shows the running time of the optimizer using each of our heuristics, as a percentage of the optimizer running time when using the full exponential search. The time for the full exponential search was 0.28, 1.52, 8.75, and 72.8 seconds for 841, 24389, 103823, and 729000 group combinations examined, respectively. For small numbers of possible



(a) (b)

Fig. 9. (a) Range query median latency versus width of range searched for SingleQuery on PlanetLab. The error bars show the 90th and 10th percentile values over 10 runs. (b) Optimizer median latency versus width of range searched, on PlanetLab. The error bars show the 90th and 10th percentile values over 10 runs.

group combinations, the benefits of using the heuristics are relatively small since the runtime for the full search is small. But for larger problems, the savings gained from using a heuristic is significant, reducing a 72 second search to a few seconds.

Reducing the time of the search is only useful if the result returned maintains an acceptable level of "accuracy," which in this case implies that the solution is found (if it exists) and it is still close to optimal. Figure 8(b) shows the accuracy of the various heuristics relative to the results that the full exponential search finds. These results show that some of the heuristics perform significantly better than others. The "3-seconds" heuristic performs well for small searches, though for half of the experiments the full exponential search completed in less than 3 seconds anyway. The "top half" heuristic performs well in all cases and, for our workload, actually finds the optimal solution in all but one case. The "first answer" approach is the least accurate heuristic, particularly in the cases where the inter-group constraint is heavily weighted, returning solutions with penalties substantially higher than the optimal penalty. The explanation for this result is that the inter-group latency is the dominant factor in the total penalty achieved, so the sorting that helps find good answers early by considering groups with lower penalties first does not have a noticeable impact. Finally, the "top 5" heuristic does not find a feasible solution at all in half of our cases, while the other approaches always do. Thus we conclude that the "top half" heuristic offers the best performance-accuracy tradeoff.

4.3 End-to-end Performance on PlanetLab

In addition to evaluating SWORD in an emulated environment, we evaluated SWORD on PlanetLab. Compared to ModelNet, PlanetLab has a smaller number of nodes, more processor contention, and a wide range of inter-node bandwidths. We ran our experiments on PlanetLab in July 2004 on two sets of nodes (one is a subset of the other): i) all 214 usable nodes that were connected to the commodity

Internet, that is, all usable nodes that were not connected only to Internet-2[2]; and ii) the subset of the first set that is used by CoDeeN, a content distribution network that runs on PlanetLab. This second set of 108 nodes are all at universities in North America and tend to have high-bandwidth, low-latency network paths to one another. We deployed SWORD in the SingleQuery configuration.

Each node reported the same number of metrics as our ModelNet experiments. 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. We issued a series of queries, each requesting two groups of 4 nodes each, such that the inter-node latency among all nodes within each group was between 0 ms and 150 ms, and the load on each node was between 0 and N, where N was varied in each query so as to cover all integers between 1 and 15, inclusive.

Figures 9(a) and (b) show the median latency for the distributed range query and the optimizer runtime, as a function of the upper bound of the load requested, and hence the number of candidate nodes returned, with a query rate of one query per minute. The number of candidate nodes returned ranged from 108 to 214 for the all-non-I2-nodes configuration, and 34 to 108 for the CoDeeN-only configuration, depending on the range of loads requested. The number of DHT nodes searched ranged from 2 to 9 and 1 to 6 for the two configurations, respectively.

Figure 9(a) 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 also shows that for these relatively small configurations, in which at most 9 nodes are searched, most performance effects are in the "noise" except for the number of candidate nodes returned. This observation suggests that *if* real-world user queries on PlanetLab commonly return hundreds of candidate nodes, we can improve SWORD's performance by returning fewer nodes at the expense of providing approximate solutions, or by compressing returned data. Figure 9(b) shows SWORD's optimizer latency increasing as the number of candidate nodes increases. Here the increasing latency is not due to data transfer, but because of the larger input to the optimizer algorithm. This graph shows that on a shared platform where other jobs are contending for the processor (PlanetLab nodes in July 2004 were often highly loaded), optimizer latency is potentially significant if a large number of candidate nodes is returned.

Finally, Figure 10 shows range query latency as a function of query rate for SWORD in three configurations: the SingleQuery configuration, a Fixed-1 configuration, in which all measurement reports and queries are sent to a single node, and a Fixed-2 configuration in which half of all measurement reports are sent to one node, half of all measurement reports are sent to a second node, and all queries are sent to both nodes. The workload for this experiment was identical to that for the "All non-I2-only nodes" configuration of the previous experiment with "load requested" set to 15. However, this experiment was run several months after the one described earlier in this section, so the absolute latencies are not directly comparable. The error bars in the graph correspond to the SingleQuery configuration; we omit er-

[2]Bamboo needs symmetric reachability among nodes, hence this restriction.

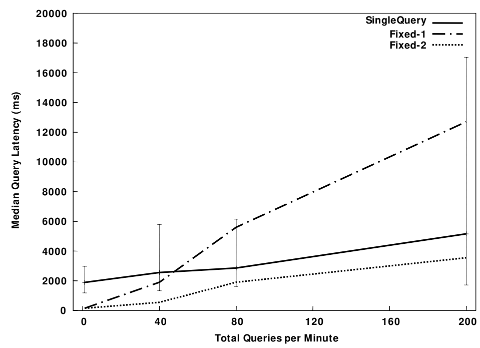


Fig. 10. Range query latency versus workload rate, expressed as total queries per minute issued across the entire system. The error bars show 10th and 90th percentile values over 10 runs.

ror bars for the centralized configurations for clarity. The important observation in this graph is that for these workloads, the Fixed-2 centralized implementation always outperforms the DHT-based implementation and the Fixed-1 centralized implementation outperforms the DHT-based implementation with a query rate at or below forty queries per minute.

5. OBSERVATIONS AND LESSONS LEARNED FROM SWORD ON PLANETLAB

In this section, we present several lessons learned from our deployment and operation of SWORD on PlanetLab for several months. For each observation, we have italicized the point we feel generalizes to services beyond SWORD and likely to future testbeds other than PlanetLab.

The claim has been made that DHTs are important distributed application building blocks because *a service built on top of a DHT will automatically inherit the DHT's self-configuration, self-healing, and scalability.* We found this claim to be largely true, although our centralized architectures did outperform the distributed designs when there was sufficient bandwidth and processing power for the central servers to operate. Using a DHT also simplified the implementation of our distributed architectures. Bamboo's neighbor heartbeat mechanism and node join bootstrap protocol automatically repartitioned the keyspace—and hence the range partition mapping of measurement reports to servers—when DHT nodes joined or left, voluntarily or due to failure or recovery, without the need for human involvement or application-level heartbeats within SWORD.

Our second observation stems from the fact that managed infrastructure distributed testbeds like PlanetLab tend to be small, particularly in contrast to end-user peer-to-peer networks like Kazaa [Kazaa 2001] or public resource computing networks like BOINC [Red Herring Magazine 2004], both of which typically have thousands to perhaps millions of nodes participating at any one time. To handle the resource discovery needs of a platform the size of PlanetLab, a resource discovery system with a few centralized server sites, each containing one or a few clustered machines, provides sufficient performance, as Figure 10 suggests.

More generally, at such small scales, simple architectures frequently outperform peer-to-peer architectures. This advantage is especially true for non-compute-intensive, low-bandwidth applications, as such applications by their nature will not benefit significantly from the many extra processors and network links that peer-to-peer architectures provide. SWORD is an example of such an application: although its performance benefits from parallelizing the work of the optimizer on a per-query basis, the dominant performance factor is the sequential visiting of nodes during the range search, with each traversal incurring one average-latency network hop. Such multiple wide-area network hops are unnecessary when the application is deployed across a small testbed. Thus, our observation is that *when deploying a system that targets a platform the size of PlanetLab, it may be worthwhile to evaluate a centralized implementation before embarking on a peer-to-peer implementation.* Among the factors that should be considered are as follows.

— *Performance for expected workload and scale:* As already mentioned, the tradeoffs between peer-to-peer and more centralized implementations are highly dependent on expected system scale and workload.

— *Availability and disaster tolerance requirements:* DHTs provide node and network link failure and recovery detection and automatic repartitioning of ownership of the keyspace in the event of node and network link failure and recovery. Such automatic failover is also possible in centralized implementations, though typically requiring an external mechanism such as a front-end failure-detecting load balancing switch. DHTs provide disaster tolerance when they replicate data; non-peer-to-peer multi-site implementations can also provide disaster tolerance at the expense of additional failover mechanisms.

— *Implementation effort, given desired features:* It is much easier to build a service to run on a single node than it is to build a service to run on a peer-to-peer network. Between these two extremes are services built to run on a cluster at a single site or across a small number of sites. The implementation effort required for these various architectures depends on the type of service implemented and its desired scalability and availability.

— *Debugging effort:* Likewise, it is much easier to debug an application running on a single node than it is to debug one running on a peer-to-peer network. Between these two extremes are services built to run on a cluster at a single site or across a small number of sites. The implementation effort required for these various architectures is application-specific.

— *Security:* Peer-to-peer networks, whether installed in users' homes or spread across a large number of managed sites as in PlanetLab, store data on hundreds or more nodes. Because of the wide distribution and federated management of these nodes, it is likely to be easier for an attacker to compromise data in a distributed implementation than it is for an attacker to compromise data when the service is deployed at fixed, pre-authenticated datacenters. We discuss specific security issues related to SWORD in Section 7.

Another challenge we overcame during our deployment of SWORD was keeping the service running continuously on PlanetLab. We used the PlanetLab Application Manager [Huebsch 2004] to automatically restart crashed SWORD instances. It was important to disable this feature during debugging, however, since in that setting a crashed application instance generally indicates a bug that needs to be fixed. *Automatic re-start was a mixed blessing* once we had deployed the service in "production." While it allowed SWORD to recover quickly from node reboots, and allowed us to continue to provide the service in the face of bugs, it hid transient bugs. Because periodically collecting logfiles from hundreds of machines to look for restarts is time-consuming and resource intensive, a more sensible approach is to automatically email the service operator the most recent logfile each time the application is restarted on a node. Restart allows a service to handle failure gracefully, but care must be taken to notify an operator and preserve forensic evidence when failures do occur, so that the underlying bug can be fixed.

Finally, we note that when it comes to evaluating distributed applications, PlanetLab offers several benefits and drawbacks compared to more traditional platforms such as fast network simulation[3] and cluster-based emulation. These tradeoffs fall into the following categories.

— *Scale:* During our measurement period, PlanetLab consisted of about 500 physical nodes, about two thirds of which were typically online and functional at any one time. In contrast, cluster-based emulation typically allows evaluation of applications with ten or twenty "virtual nodes" per physical cluster node; for example, when evaluating SWORD, we found that we could emulate a network of about 1000 "virtual nodes" on 40 physical nodes before processor contention due to multiplexing many virtual nodes per physical machine began to affect our performance measurements. Simulation is arguably the most scalable evaluation method. Unlike PlanetLab and emulation, it does not run in real-time, so an arbitrarily large system can be simulated if the experimenter is willing to wait long enough and is able to provision the machine running the simulation with sufficient virtual memory.

— *Network topology and link characteristics:* The PlanetLab network topology is "hard-wired" based on the nodes that have elected to participate in PlanetLab, but these network links reflect real-world latency, bandwidth constraints, and packet loss rates. In contrast, simulation allows the experimenter to create an arbitrary network topology, but fast network simulators typically simulate only network latency, not bandwidth constraints or packet loss. Emulation offers the best of both words, enabling the creation of arbitrary network topologies and emulation of user-defined latency, bandwidth, and loss rate on each network link.

— *Competition for resources:* Applications running on PlanetLab nodes are exposed to contention from other applications for processor, memory, disk, and network resources. Applications running in emulation can be subjected to contention for these resources in a more controlled way. Fast network simulators, because they are network simulators and not processor or operating system simulators, cannot evaluate the application impact of contention for node resources. Thus emulation, and to some extent PlanetLab deployment, are the best platforms for studying application sensitivity to resource constraints.

— *Workload:* A key benefit of PlanetLab deployment is that applications can be offered as a service to others, and thereby can be exposed to a real user workload.

[3]By "fast network simulation" we mean simulators such as p2psim [Li et al. 2005] or the simulator that is distributed with Bamboo and Tapestry.

Of course, the challenge is attracting enough users to the service to meaningfully characterize typical usage. Our current SWORD deployment has not yet met this criteria, but we hope that its integration with Plush, described in Section 3.3, will attract more users and thereby enable to us to draw conclusions about what a "typical" SWORD query workload is. On the other hand, we *were* able to use PlanetLab to derive a typical SWORD "update" workload, by observing the node utilization data that SWORD measured from Ganglia and other sources.

— *Operator actions:* Because PlanetLab nodes are deployed across hundreds of sites and operated by local site administrators and PlanetLab Central, they are subjected to a realistic set of operator actions—some beneficial to availability and/or performance, and some detrimental. However, users cannot subject nodes to the full range of operator actions that they can in a dedicated cluster environment, because they do not have access to the physical machine or root access to the underlying operating system[4]. Finally, operator actions cannot be realistically represented at all in a fast network simulator, as neither the hardware nor operating system is simulated.

— *Faults:* Because PlanetLab nodes are deployed across hundreds of sites, they are subjected to a realistic set of hardware, operating system, and operator faults. Furthermore, application deployers are free to inject application-level faults of their choosing. However, because PlanetLab users do not have access to the hardware or underlying operating system, they are limited to injecting arbitrary faults into the application and virtualized operating system (vserver) layer of their system only. Emulation, because it runs in a dedicated cluster environment, allows introduction of the full set of hardware, operating system, and application-level faults. Simulation cannot realistically reflect faults; in essence, the best a simulator can do is to map all faults into a single fault, namely node disconnection from the simulated network.

— *Reproducibility:* The large size of the error bars in our graphs in Figure 10 shows that one of the key drawbacks of PlanetLab experimentation is reproducibility. Due to varying contention for processor, memory, network, and disk I/O resources, performance varies significantly over even short time periods. This fact makes it difficult to use PlanetLab to ascertain the performance impact of a design parameter by varying that parameter over multiple runs, as many other factors will be varying across those runs outside of the experimenter's control. In contrast, the controlled environments of an emulation cluster and a simulator allow the same experiment to be repeated multiple times, with only the desired parameter(s) changing between runs. We note that simulation is slightly more reproducible than emulation because it allows a repeatably deterministic ordering of events, whereas emulation allows events to become arbitrarily interleaved among different nodes.

— *Experiment management:* Deploying an application on a single-node simulator is generally easy. Deploying an application on a cluster is slightly more difficult

[4]Each PlanetLab slice (user) is given a Linux vserver [Linux VServer 2003] environment on each machine, providing namespace isolation (for files, processes, and other entities) but not a true virtual or physical machine. This allows users to perform some of their own operator actions on PlanetLab, but not the full set possible on a truly dedicated machine.

because code and data must be distributed to the cluster nodes, the application must be started and stopped on the various nodes, and relevant logfiles must be copied back to a central node if centralized analysis is desired. The same steps must be taken to run an application on PlanetLab, but with an additional twist: nodes and network links may fail and recover during the application's deployment and operational phases. Tools that deploy and monitor the application—not just the application itself—must therefore handle failures gracefully. Moreover, to understand application behavior, an experimenter must "factor out" the failures that occurred outside of their control as the experiment ran.

Based on these observations, we conclude that PlanetLab evaluation *complements*, rather than *replaces*, traditional evaluation approaches. We believe that in the future, it will be common for system developers to obtain the "best of both worlds" by deploying their system on PlanetLab to obtain traces and/or models of realistic workload, contention, and failures, and then using those traces and/or models to drive simulation or emulation. In this study, we have started along these lines in a small way by generating the SWORD "update workload" for our experiments from a measurement trace taken from PlanetLab.

6. RELATED WORK

SWORD builds on work in resource discovery, Internet-scale query processing, and distributed range search. In this section we discuss how SWORD compares to related work in each of these areas.

6.1 Resource discovery

Resource discovery has long been a research topic in the Grid community [Czajkowski et al. 2001]. The most widely-deployed Grid resource discovery system is the Globus Toolkit Monitoring and Discovery Service (MDS2) [Zhang and Schopf 2004]. MDS2 defines an architecture in which "Information Providers" provide raw measurement data, a "Grid Resource Information Service" (GRIS) makes the information available for querying, and a "Grid Index Information Service" (GIIS) aggregates data from GRISes. Globus provides an OpenLDAP-based per-node GRIS and allows users to plug in their own GIIS implementations. MDS2 data aggregation follows an administrator-specified hierarchical structure. SWORD's query processor could be used as a GIIS, connecting GRISes in a peer-to-peer fashion. MDS3 and MDS4 have recently emerged as successors to MDS2 with similar goals.

Kee *et al.* describe "virtual grids" [Kee et al. 2005]. The description language vgDL allows users to describe resource requirements as hierarchies of homogeneous or heterogeneous groups of nodes with good or poor connectivity, reminiscent of SWORD's groups with per-node, inter-node, and inter-group constraints, but with coarser-grained specifications and support for arbitrarily deep hierarchies. The resource mapping component vgFAB stores resource measurements only in a centralized database, in contrast to SWORD which has the ability to store measurements in groups of central servers or in a DHT. Also, vgFAB computes a bounded set of pre-fabricated groups and stores them in the database rather than dynamically forming them on a query-by-query basis like SWORD.

Condor and its ClassAds language [Litzkow et al. 1988] provide similar functionality to virtual grids and SWORD, absent the notion of groups and inter-node connectivity constraints. Gang matching [Raman et al. 2003] extends Condor's original bilateral matching scheme to a multilateral one, allowing co-allocation of multiple resources. Set matching [Liu et al. 2002] allows requests that express aggregate constraints. SWORD offers ClassAds as one of its query languages; this provides users the flexibility of ClassAds' more general semantics for ranking candidate nodes, but using this syntax users cannot specify inter-node or inter-group constraints or ranking functions. Redline [Liu and Foster 2004] formulates the matching problem as a constraint satisfaction problem. These latter systems allow expression of resource groups, but they do not offer a concise method to express network topologies. Also, to date their implementations have been centralized.

XenoSearch [Spence and Harris 2003] supports DHT-based multi-attribute range queries in a manner similar to our MultiQuery approach, but it uses a separate DHT instance per attribute, creates its query routing structure explicitly rather than using built-in DHT successor pointers, and provides approximate answers using Bloom Filters. Additionally, SWORD allows users to define groups with inter-node and inter-group requirements and "penalty functions" to rank nodes meeting the requirements. SWORD penalty functions are a simple version of more general utility functions, which are well studied components in many resource management and allocation mechanisms. Though utility functions are often discussed in the context of microeconomics and game theory, [Ferguson et al. 1996] and [Ibaraki and Katoh 1988] provide a survey of some of the earlier computer science research that focuses on utility functions in resource allocation schemes.

Huang and Steenkiste [Huang and Steenkiste 2003] describe a mechanism for network-sensitive service selection. Their system addresses problems similar to the ones that SWORD addresses, but using only centralized data collection and resource mapping. They focus on finding single groups that meet target criteria for a desired application, rather than multiple groups with inter-node and inter-group characteristics as SWORD does.

The network topology embedding problem is formulated as a constraint satisfaction problem in [Considine et al. 2003] for wide-area networks and as an optimization problem in [White et al. 2002] for cluster networks. Similar to SWORD, the former system finds the best "embedding" of a user's requested emulated network topology within the graph of PlanetLab nodes, while the latter maps a user's requested emulated network topology to a set of physical cluster resources that offer emulation capabilities.

PlaceLab [Chawathe et al. 2005] is a resource discovery service for locating nearby WiFi hotspots. Organizations store a <latitude, longitude> tuple corresponding to each hotspot they operate, and arbitrary users may also store such location information when they discover a hotspot whose administrators have not explicitly registered the hotspot with PlaceLab. Users wishing to locate a hotspot near them may query PlaceLab with their current location, and the infrastructure returns a list of nearby hotspots.

Other Internet technologies, including the Service Location Protocol (SLP) [SLP 1987], Domain Name System (DNS) [DNS 1987], and Lightweight Directory Access Protocol (LDAP) [LDAP 1997] have been in use for many years and have similar goals as SWORD. SLP is a protocol that allows applications to discover networked services without knowing the specific hostname or IP address of the desired resource. Instead, applications simply describe the desired service and SLP finds the URLs for the appropriate resources. Jini [Jini 1998] is an example of a specific Java technology that provides this service. DNS is an Internet technology that is used to translate alphabetic domain names into IP addresses. This prevents users from having to remember IP addresses for resources they wish to access, since domain names like "www.google.com" are easier to remember than strings of numbers. Similarly, LDAP is a protocol that is used to access a directory that stores the location of organizations, individuals, and other networked devices. Thus a user who wishes to access a particular resource uses LDAP to look up the resource in a directory. Like SWORD, in all three of these instances, users benefit from not needing to know all information about all available resources ahead of time.

6.2 Internet-scale query processing

PIER [Huebsch et al. 2003], Sophia [Wawrzoniak et al. 2003], IrisNet [Nath et al. 2003], and Astrolabe [van Renesse et al. 2003] provide Internet-scale query processing. All four could be used to satisfy per-node resource queries, and they offer a more expressive language for specifying such requirements than SWORD. However, the first three must contact all data-storing nodes to perform range search and the last disseminates measurement data globally, while SWORD targets its range search to only the nodes storing measurements within the target attribute's range. Also, SWORD provides a query language and optimizer specialized for specifying and optimizing groups of nodes with inter-node requirements and preferences. We note that PIER implements an efficient distributed join primitive that SWORD does not, which could allow faster processing of range queries over inter-node characteristics.

6.3 Distributed range search

DHT-based range search was suggested initially by Karger and Ruhl [Karger and Ruhl 2004], using the technique of mapping values of an attribute to DHT keys. They also suggested an "item balancing" algorithm that reassigns Node IDs so as to spread load evenly among nodes in the face of a non-uniform workload, though at the expense of path length inflation. Mercury [Bharambe et al. 2004] extends this work to multi-attribute queries using the concept of *hubs*, similar to our *subregions*. They also describe a small-world based routing overlay that restores logarithmic routing path length. Mercury is evaluated primarily via simulation, with a focus on routing efficiency and load balance. In contrast, SWORD focuses on end-to-end performance and resource consumption, in emulation and a live deployment, when multi-attribute range search is applied to resource discovery. Additionally, we describe passive load balancing that complements Mercury's active load balancing.

Another recent proposal for distributed range search is the Prefix Hash Tree (PHT) [Ramabhadran et al. 2004]. PHT is designed to perform single-dimensional range queries on top of DHTs while maintaining load balance. PHT incrementally grows a trie, where each leaf 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, namely the root, and grows by incrementally splitting nodes when they become overloaded. The

PlaceLab mapping infrastructure described earlier uses the PHT algorithm running atop OpenDHT [Rhea et al. 2005]. It converts <latitude, longitude> tuples into a single-dimensional attribute using a Z-curve linearization.

Researchers have also recently proposed distributed range search using skip graphs [Aspnes and Shah 2003; Aspnes et al. 2004; Awerbuch and Scheidler 2003], and using special-purpose data structures built on top of CAN [Ratnasamy et al. 2001; Tang et al. 2003] and Chord [Stoica et al. 2001; Gupta et al. 2003; Crainiceanu et al. 2004].

7. CONCLUSION AND FUTURE WORK

In this paper we have described SWORD, a scalable resource discovery service for wide-area distributed systems. SWORD users describe a requested system topology in terms of groups with required intra-group, inter-group, and per-node characteristics whose relative importance and sensitivity are expressed using penalty functions. We explore a number of distributed and centralized query algorithms for finding nodes meeting required per-node constraints, and several heuristics for finding the best mapping of nodes to groups. Through emulation-based evaluation, we find that an architecture based on 4-node server clusters at network peering facilities outperforms a DHT-based resource discovery infrastructure for all but the smallest number of sites.

Our experience with SWORD's PlanetLab deployment shows that this DHT-based approach performs reasonably well despite competition from numerous processor and network-intensive applications sharing the same PlanetLab infrastructure nodes. While our results are specific to the system architectures and workload configurations we examined, we believe that our experience considering both centralized and distributed architectures provides interesting guidelines and insights regarding appropriate architectures for a variety of systems depending on available resources, expected level of load, and required levels of performance and availability. In operating a live deployment of SWORD on PlanetLab for more than a year, we found that SWORD benefitted significantly from the DHT's self-healing properties but less from its scalability properties due to the small size of the platform. As an evaluation platform, we found that PlanetLab is an excellent addition to, but not a replacement for, emulation. In particular, PlanetLab does not allow creation of arbitrary network topologies or injection of node-level faults, and results on the platform are less reproducible than those from emulation, due to unpredictable competition with other PlanetLab users for node and network resources.

An important area of future work is security. Nodes could sign measurement reports and queries as a form of authentication. Given an authentication infrastructure, per-node rate limiting could ensure that no node utilizes more than a predefined amount of bandwidth (or optimizer CPU time) per unit time on any single node. Such a technique is vulnerable to the Sybil attack [Douceur 2002] and therefore requires a trusted identity creation service. To ensure that nodes are truthful in their measurement reports, a verification service could run microbenchmarks to verify that resource availability matches earlier advertisements. To ensure that, modulo collusion, nodes are truthful when they run the optimizer, a client might issue each query to several query nodes and compare the results.

Privacy is another challenging security issue for distributed versions of SWORD. Reporting nodes could encrypt attribute names to hide their meanings, but our range search mechanism relies on a monotonic mapping function from measured values to DHT keys, and encrypting values using standard techniques, either before or after mapping them to a DHT key, will break this monotonicity.

Finally, we have not yet studied the system dynamics that result from multiple large-scale applications periodically querying SWORD to determine when and how to migrate application instances. We anticipate that mechanisms are needed to dampen potential oscillations. Further, providing support for more complex penalty functions is another extension we hope to explore in the future.

More information on SWORD including the PlanetLab deployment can be accessed at http://www.swordrd.org/.

REFERENCES

Albrecht, J., Tuttle, C., Snoeren, A. C., and Vahdat, A. 2006. PlanetLab Application Management Using Plush. *SIGOPS-OSR 40,* 1, 33–40.

Aspnes, J., Kirsch, J., and Krishnamurthy, A. 2004. Load Balancing and Locality in Range-queriable Data Structures. In *Proceedings of PODC.*

Aspnes, J. and Shah, G. 2003. Skip Graphs. In *Proceedings of SODA.*

AuYoung, A., Chun, B. N., Snoeren, A. C., and Vahdat, A. 2004. Resource Allocation in Federated Distributed Computing Infrastructures. In *Proceedings of OASIS.*

Awerbuch, B. and Scheidler, C. 2003. Peer-to-Peer Systems for Prefix Search. In *Proceedings of PODC.*

Balazinska, M., Balakrishnan, H., and Karger, D. 2002. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of ICPC.*

Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., and Wawrzoniak, M. 2004. Operating Systems Support for Planetary-Scale Network Services. In *Proceedings of NSDI.*

Bharambe, A., Agrawal, M., and Seshan, S. 2004. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *Proceedings of SIGCOMM.*

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

Chawathe, Y., Ramabhadran, S., Ratnasamy, S., LaMarca, A., Shenker, S., and Hellerstein, J. 2005. A Case Study in Building Layered DHT Applications. In *Proceedings of SIGCOMM.*

Chen, Y., Bindel, D., Song, H., and Katz, R. 2004. An Algebraic Approach to Practical and Scalable Overlay Network Monitoring. In *Proceedings of SIGCOMM.*

Chun, B. Slicestat. http://berkeley.intel-research.net/bnc/slicestat/.

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

Crainiceanu, A., Linga, P., Gehrke, J., and Shanmugasundaram, J. 2004. Querying Peer-to-Peer Networks Using P-trees. In *Proceedings of WebDB.*

Czajkowski, K., Fitzgerald, S., Foster, I., and Kesselman, C. 2001. Grid Information Services for Distributed Resource Sharing. In *Proceedings of HPDC.*

Czajkowski, K., Foster, I., Kesselman, C., Sander, V., and Tuecke, S. 2002. SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems. In *LNCS.* Number 2537. 153–183.

Dabek, F., Cox, R., Kaahoek, F., and Morris, R. 2004. Vivaldi: A Decentralized Network Coordinate System. In *Proceedings of SIGCOMM.*

Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., and Stoica, I. 2003. Towards a Common API for Structured P2P Overlays. In *Proceedings of IPTPS.*

DNS 1987. http://www.ietf.org/rfc/rfc1035.txt.

Douceur, J. R. 2002. The Sybil Attack. In *Proceedings of IPTPS*.

Ferguson, D., Nikolaou, C., Sairamesh, J., and Yemini, Y. 1996. Economic Models for Allocating Resources in Computer Systems. World Scientific (Editor: Scott Clearwater).

Foster, I. and Kesselman, C. 2003. *The Grid 2*. Morgan Kaufmann.

Foster, I., Kesselman, C., and Tuecke, S. 2001. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *IJSA 15,* 3.

Fu, Y., Chase, J., Chun, B., Schwab, S., and Vahdat, A. 2003. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of SOSP*.

Gupta, A., Agrawal, D., and Abbad, A. E. 2003. Approximate Range Selection Queries in Peer-to-Peer Systems. In *Proceedings of CIDR*.

Huang, A. and Steenkiste, P. 2003. Network-Sensitive Service Discovery. In *Proceedings of USITS*.

Huebsch, R. 2004. Planetlab application manager. http://appmanager.berkeley.intel-research.net/.

Huebsch, R., Hellerstein, J. M., Boon, N. L., Loo, T., Shenker, S., and Stoica, I. 2003. Querying the Internet with PIER. In *Proceedings of VLDB*.

Ibaraki, T. and Katoh, N. 1988. Resource Allocation Problems: Algorithmic Approaches. MIT Press, Cambridge, MA.

Jagadish, H. V. 1990. Linear Clustering of Objects with Multiple Attributes. In *Proceedings SIGMOD*.

Jini 1998. http://java.sun.com/products/jini.

Karger, D. and Ruhl, M. 2004. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proceedings of IPTPS*.

Kazaa 2001. http://www.kazaa.com/us/index.htm.

Kee, Y.-S., Logothetis, D., Huang, R., Casanova, H., and Chien, A. 2005. Efficient Resource Description and High Quality Selection for Virtual Grids. In *Proceedings of CCGrid*.

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

LDAP 1997. LDAP. http://www.ietf.org/rfc/rfc2251.txt.

Li, J., Stribling, J., Morris, R., Kaashoek, M. F., and Gil, T. M. 2005. A Performance vs. Cost Framework for Evaluating DHT Design Tradeoffs Under Churn. In *Proceedings of INFOCOM*.

Linux VServer 2003. http://linux-vserver.org/.

Litzkow, M., Livny, M., and Mutka, M. 1988. Condor – A Hunter of Idle Workstations. In *Proceedings of ICDCS*.

Liu, C. and Foster, I. 2004. A Constraint Language Approach to Matchmaking. In *RIDE*.

Liu, C., Yang, L., Foster, I., and Angulo, D. 2002. Design and Evaluation of a Resource Selection Framework. In *Proceedings of HPDC*.

Massie, M., Chun, B., and Culler, D. 2004. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing 30,* 7 (July).

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

Ng, T. S. E. and Zhang, H. 2002. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proceedings of INFOCOM*.

Ng, T. S. E. and Zhang, H. 2004. A Network Positioning System for the Internet. In *Proceedings of USENIX ATC*.

Oppenheimer, D., Chun, B., Patterson, D., Snoeren, A. C., and Vahdat, A. 2006. Service Placement in Shared Wide-Area Platforms. In *Proceedings of USENIX ATC*.

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

Pai, V. S., Wang, L., Park, K., Pang, R., and Peterson, L. 2003. The Dark Side of the Web An Open Proxy's View. In *Proceedings of HotNets*.

Ramabhadran, S., Ratnasamy, S., Hellerstein, J. M., and Shenker, S. 2004. Prefix Hash Tree. In *Proceedings of PODC*.

Raman, R., Livny, M., and Solomon, M. 1998. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of HPDC*.

Raman, R., Livny, M., and Solomon, M. 2003. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In *Proceedings of HPDC*.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. 2001. A Content Addressable Network. In *Proceedings of SIGCOMM*.

Red Herring Magazine 2004. Distributed computing: We come in peace. august 31, 2004.

Reynolds, P. and Vahdat, A. 2003. Efficient Peer-to-Peer Keyword Searching. In *Proceedings of Middleware*.

Rhea, S., Chun, B.-G., Kubiatowicz, J., and Shenker, S. 2005. Fixing the Embarrassing Slowness of OpenDHT on PlanetLab. In *Proceedings of WORLDS*.

Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. 2004. Handling Churn in a DHT. In *Proceedings of USENIX ATC*.

Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., and Yu, H. 2005. OpenDHT: A Public DHT Service and its Uses. In *Proceedings of SIGCOMM*.

SLP 1987. http://www.ietf.org/rfc/rfc2165.txt.

Spence, D. and Harris, T. 2003. XenoSearch: Distributed Resource Discovery in the XenoServer Open Platform. In *Proceedings of HPDC*.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM*.

Tang, C. and Dwarkadas, S. 2004. Hybrid Global-Local Indexing for Efficient Peer-to-Peer Information Retrieval. In *Proceedings of NSDI*.

Tang, C., Xu, Z., and Mahalingam, M. 2003. pSearch: Information Retrieval in Structured Overlays. *SIGCOMM CCR 33,* 1, 89–94.

Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostić, D., Chase, J., and Becker, D. 2002. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of OSDI*.

van Renesse, R., Birman, K., and Vogels, W. 2003. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM TOCS 21,* 2, 164–206.

Wawrzoniak, M., Peterson, L., and Roscoe, T. 2003. Sophia: An Information Plane for Networked Systems. In *Proceedings of HotNets*.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of OSDI*.

Zhang, X. and Schopf, J. 2004. Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2. In *Proceedings of MP*.