The XtreemOS Resource Selection Service

CORINA STRATAN, VU University Amsterdam
JAN SACHA and JEFF NAPPER, Bell Labs
PAOLO COSTA, Imperial College London
GUILLAUME PIERRE, VU University Amsterdam

Many large-scale utility computing infrastructures comprise heterogeneous hardware and software resources. This raises the need for scalable resource selection services that identify resources that match application requirements. Such a service must provide an efficient lookup in spite of changing resource attributes such as disk size, changing application requirements such as installed software libraries, and changing system composition as resources join or leave. We present a fully decentralized, self-managing Resource Selection Service (RSS) algorithm by which resources autonomously select themselves when their attributes match a query. An application specifies what it expects from a resource by means of a conjunction of (attribute,value-range) pairs, which are matched against the attribute values of resources. The set of search attributes can also be updated online to reflect new requirements. We show that our solution scales in the number of resources and in the number of attributes, while being relatively insensitive to churn and other membership changes like node failures. Our RSS continuously self-adapts its routing structure in response to variations in the distribution of node attributes and queries. We show that this autonomous optimization maintains performance and availability in a long-lived service even when the set of application requirements used to select resources changes.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of systems**]: Design studies; H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms: Performance, Reliability

Additional Key Words and Phrases: XtreemOS, RSS, resource selection, self-adaptation.

1. INTRODUCTION

Applications are increasingly voracious in the computing resources they require to execute efficiently. Moreover, the computing demand of many users changes over time. Instead of statically assigning resources to applications, we observe a steady growth in the adoption of the utility computing model in large-scale systems like Grids, Peer-to-Peer systems and Clouds. According to this model, the execution of applications is outsourced to a (potentially very large) shared external infrastructure of compute and storage resources that can manage both long-term growth and short-term fluctuations in use.

In such large-scale systems, the capabilities of individual resources are often very diverse [Anderson and Reed 2009]. Therefore, an essential instrument is a resource selection service capable of identifying suitable resources for executing each application. The resource selection service should provide a *lookup* primitive that takes a specification of required resource attributes and returns a list of machines suitable for running the concerned application.

The research leading to these results has received funding from the European Union's Sixth Framework Programme under grant agreement IST-FP6-033576.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Although previous research work targets the problem of resource selection in distributed systems, some challenges related to this problem have not been completely addressed yet. An essential issue is the scalability, both in terms of the number of nodes the service can support and in the supported number of node attributes; utility computing platforms might need to maintain large numbers of attributes per node to represent hardware characteristics as well as the (non-)availability of certain libraries or other administrative properties.

Besides scalability, another challenge in large scale distributed systems is their dynamic and complex behavior: computing resources can join or leave the system at any time, creating so-called *churn;* whole data centers might become unreachable due to network failures; operating systems and software systems might be upgraded or modified; and application requirements might change over time. To maintain efficient behavior, the system needs to dynamically self-adapt to all such changes. While some self-adaptation may occur independently at each node, other changes require coordination among a vast majority of nodes to maintain the system's correctness in the presence of churn. For example, nodes can locally perform maintenance of the overlay by periodically checking for failed neighbors. However, changing the attribute space to add or remove search attributes requires coordination among nodes to prevent conflicting views of the search space. Currently, such changes requiring global coordination are done offline by administrators and are not amenable to online dynamic control.

The resource selection solutions proposed previously range from centralized and hierarchical node directories to DHT-based solutions [Ranjan et al. 2008]. Most of these approaches rely on delegation, where compute nodes register their attributes to registry nodes that implement the lookup functionality. Registry nodes must then monitor the availability of compute nodes and periodically refresh the registered attribute values to maintain accuracy. We claim that delegation should be avoided for four reasons: (i) it creates unnecessary load on the system due to the periodic revalidations of the registered values and the need to check node availability regularly; (ii) it creates inconsistency between the actual and registered attribute values, for example, in the case of a failure of a compute node or its corresponding registry node(s); (iii) it creates imbalanced workloads, requiring extra effort to balance. DHT-based resource selection systems frequently divide the searchable space on a per-attribute basis and each peer in the system is then responsible for keeping references to the nodes in charge of those specific attribute values. This, however, generates uneven load distribution when a particular attribute range becomes popular; (iv) it leads to scalability problems: centralized and hierarchical registries can in principle handle any number of attributes but have limited scalability in the number of nodes, particularly in a dynamic environment that is exposed to failures, where managing a robust node hierarchy is far from trivial [van Steen and Ballintijn 2002]. On the other hand, DHT-based solutions usually scale very well with large numbers of nodes, but they do not efficiently support multidimensional-range searches.

This paper extensively discusses the Resource Selection Service (RSS) implemented in the context of the XtreemOS Grid operating system [Coppola et al. 2008]. RSS is a fully decentralized system where each compute node is solely responsible for its own attributes and self-organization is achieved through a gossip-based overlay comprising all the nodes. Given the fact that in our system each computing resource acts as a node in the service's overlay, we will use the terms *resource* and *node* interchangeably.

Self-adaptation mechanisms enable RSS to automatically maintain performance in the presence of churn, of changes in the query workload patterns and of changes in the distribution of node attribute values. These adaptation mechanisms enable good performance even during online changes in the set of attribute values.

In RSS nodes are directly responsible for providing accurate and timely information about their resources, and to minimize overhead queries are routed quickly to nodes that can provide the desired resources. Each node is conceptually placed in a multi-dimensional

space where each dimension represents a resource-attribute type. A query is specified as a list of (*attribute*, *value interval*) pairs, effectively demarcating a subregion in this multi-dimensional space. Irrelevant attributes for a given job can be left unspecified. Nodes maintain a few links to other nodes in an overlay network so that queries can be forwarded to a node that lies either in the associated subregion or is closer to that region. Once a query is being processed within the associated subregion, it needs merely to be forwarded to enough nodes within that subregion to satisfy the query. Gossip-based epidemic protocols [Voulgaris and van Steen 2005] manage the overlay network, enabling high resilience to faults with negligible overhead. A gossip-based aggregation protocol [Sacha et al. 2010] is used to continuously estimate the statistical distribution of node attribute values and of query ranges. Based on these estimations, RSS self-adapts the way the multi-dimensional space is divided to minimize the routing overhead. This self-adaptation maintains routing efficiency for queries even in the presence of large variations in the attribute values of resources or the composition of queries. We are not aware of any other resource selection system that can autonomically adapt to the dynamic composition of the attribute search space, the distribution of attribute values, and the changing query workload.

This article is an extended version of a previous conference paper [Costa et al. 2009]. The additional contributions of this article are as follows: (i) we present and evaluate self-adaptation algorithms that enable RSS to maintain high performance despite changes in the query workload patterns and node attribute distributions; (ii) we present and evaluate algorithms that allow system administrators to control the set of attributes considered by the RSS without the need to stop the system; and (iii) we significantly extend the discussion on the gossip-based overlay maintenance and present new algorithms that improve convergence times.

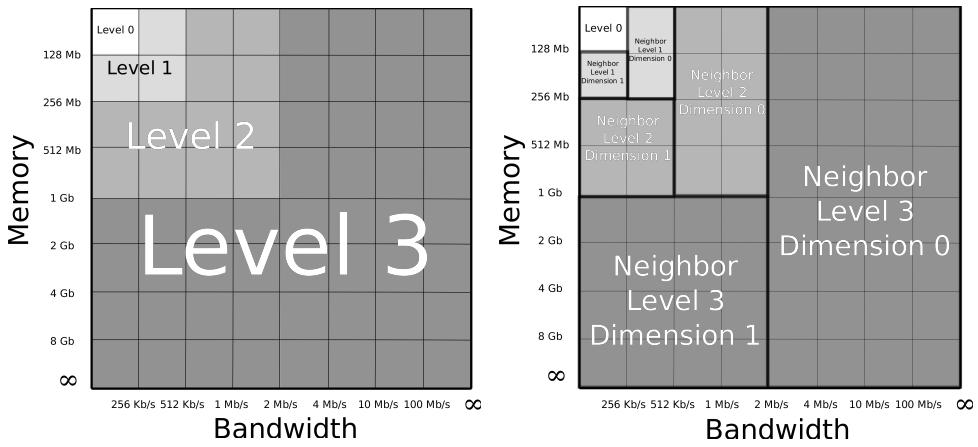
The paper is organized as follows: Section 2 presents the RSS system model. Then, Section 3 details the query routing protocol, and Section 4 shows how the overlay is constructed and maintained in the presence of churn. Section 5 details the administration functionality to change the attribute set, while Section 6 demonstrates the benefits and importance of self-adaptation to attribute value and query load distributions. Section 7 discusses related work, and Section 8 concludes the paper.

2. SYSTEM MODEL

In our model, each node is characterized by a set of (*attribute*,*value*) pairs such as memory, bandwidth and CPU power. For the sake of simplicity, in this section we assume that the number of attributes is fixed and known *a priori*. In Section 5 we discuss adding and removing attributes online.

We represent the overlay as a $d-$dimensional space $\mathcal{A} \triangleq \mathcal{A}_1 \times \mathcal{A}_2 \times \cdots \times \mathcal{A}_d$, with $\mathcal{A}_i$ being the set of all possible values for attribute a_i and d the number of different attributes considered. Every node X can therefore be represented as a single point with coordinates $(v_1, v_2, \ldots, v_d)$ where v_i is the value of attribute a_i for node X. A query is defined as a binary relation over $\mathcal{A}$, i.e., $q : \mathcal{A} \rightarrow \{0, 1\}$ that selects nodes which satisfy the application requirements. For simplicity, we focus on range queries so the query can be represented as a range [min, max] such that only nodes within the range satisfy the query. More complex queries can be represented as the union of different range queries. The set of nodes for which q yields 1 represents the set of *candidates* to be allocated to the application. Note that q identifies a subset $\mathcal{Q}(q) \triangleq \mathcal{Q}_1 \times \mathcal{Q}_2 \times \cdots \times \mathcal{Q}_d$, where $\mathcal{Q}_i \subseteq \mathcal{A}_i$.

As an example, consider a space based on five attributes: CPU instruction set, memory size, bandwidth, disk space, and operating system. An example query could then be formulated as:



(a) Nested Cells. (b) Neighboring Cells

Fig. 1. Attribute space partition with $d = 2$.

CPU	=	amd64
MEM	$\in$	$[4\,GB, \infty)$
BANDWIDTH	$\in$	$[512\,Kb/s, \infty)$
DISK	$\in$	$[128\,GB, \infty)$
OS	$\in$	{Linux 2.6.19-1.2895, ..., Linux 2.6.20-1.2944}

A query can be issued at any node; there is no designated node where queries should initially be sent to. Finally, we assume that the underlying network is fully connected such that each node can reach any other node, as it is common on the Internet.

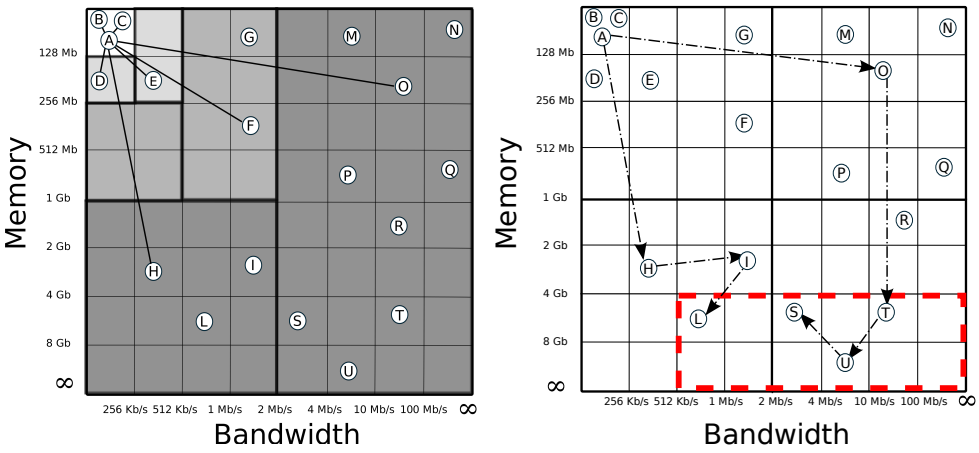
3. QUERY ROUTING

This section discusses the base resource discovery protocol without self-adaptation. We first describe the properties of the simple overlay, then detail query routing, building up an understanding of why and how RSS self-adapts to large changes in the query distribution and values of node attributes. Query routing in RSS can be optimized when the distribution of node attribute values and query ranges are known. Learning and leveraging the distribution of such values is the goal of the self adaptation mechanisms that are described in Section 6.

3.1. Overlay Network Topology

The model of node attributes described in Section 2 is naturally represented as a multi-dimensional cube. In order to scale up to large numbers of nodes, we must limit the amount of links that each node needs to maintain. A naive, inefficient solution is to connect every node, for each dimension, with its most immediate neighbors, i.e., the nodes having the most similar attribute values. When a node receives a query message q, it can then forward it in a greedy fashion to the neighbor closest to the area $\mathcal{Q}(q)$. Unfortunately, this approach creates dramatic latency and traffic overheads: since a query can be issued at any node, it may need to traverse many nodes along every dimension to reach the area $\mathcal{Q}$.

We instead opt for a hierarchical approach by recursively splitting the d-dimensional space into smaller spaces, called *cells*, and providing each node with a link to the increasingly larger subspaces of which it is a member. An example for $d = 2$ is shown in Figure 1(a). The largest cell has been partitioned into four smaller cells which each, in turn, have been split into four even smaller cells. Note that the attribute ranges of each cell do not have to be regular: One cell may range over memory between 0 and 128 MB, and another one



(a) Neighbor links for node A.

(b) Query routing from node A

Fig. 2. Two-dimensional RSS system example. The dotted rectangle indicates a query area.

between 4 GB and 8 GB. This allows us to deal with skewed distributions of attributes values. For the same reason, we do not impose an upper bound on attribute values: in our example, all nodes with more than 8 GB of RAM will be placed in the lowest row of the grid.

To distinguish among cells, we introduce the notion of level l. The smallest cells are at level zero. These are denoted as $\mathcal{C}_0$. $\mathcal{C}_1$ cells are obtained by grouping four $\mathcal{C}_0$ cells. Similarly, four $\mathcal{C}_1$ cells create a single $\mathcal{C}_2$ cell and so on. More formally, given a cube of d dimensions and a level l, a $\mathcal{C}_l$ cell is obtained by joining 2^d adjacent $\mathcal{C}_{l-1}$ cells. Every node X belongs to a unique $\mathcal{C}_l$ cell, denoted $\mathcal{C}_l(X)$.

The key to our approach is that when a node X is requested to handle a query q, X forwards the query to the lowest level cell $\mathcal{C}_l(X)$ that overlaps with $\mathcal{Q}(q)$. This approach requires that for each level l, X knows about nodes in $\mathcal{C}_l(X) \setminus \mathcal{C}_{l-1}(X)$. We construct for each dimension a neighboring subcell of $\mathcal{C}_{l-1}(X)$ by first splitting $\mathcal{C}_l(X)$ along dimension #0. The half in which $\mathcal{C}_{l-1}(X)$ is contained, is then split along dimension #1. This procedure is repeated until all dimensions have been considered, so that we will then have created d subcells at level l of $\mathcal{C}_l(X)$, each of which is adjacent to one "side" of $\mathcal{C}_{l-1}(X)$. Figure 1(b) shows the neighboring cells for a node A with the corresponding levels and dimensions.

We require that a node knows one other *neighbor* node falling in one of these subcells for each level $l > 0$, and a set of *neighbors* for the subcell with the level $l = 0$. If no node is present in a given subcell, then no link must be maintained. The nodes in $\mathcal{C}_0(X)$ are arranged in such a way that X can efficiently broadcast a message to each of them, for example through an epidemic protocol [Jelasity et al. 2007]. Links need not be symmetric. We denote the neighboring cell of node X at level l and dimension k as $\mathcal{N}_{(l,k)}(X)$. Similarly, the selected neighbor in $\mathcal{N}_{(l,k)}(X)$ is denoted as $n_{(l,k)}(X)$. Notably, while the number of $\mathcal{C}_l$ cells grows exponentially with the number of dimensions, the number of $\mathcal{N}_{(l,k)}$ subcells (and hence the number of neighbors required per node) grows only linearly, and will thus not hinder scalability.

Figure 2(a) shows an example for node A (for the sake of clarity, we omit the connections among the other nodes). First, A is connected with some other nodes in $\mathcal{C}_0(A)$ i.e., B and C. Then, for each neighboring cell $\mathcal{N}_{(l,k)}(A)$ depicted in Figure 1(b), it must choose one node $n_{(l,k)}$ to connect with. For $l = 1$, it has chosen nodes D ($k = 1$) and E ($k = 0$). For

$l = 2$, it has two available nodes for $k = 0$ (F is selected). There is no node in $\mathcal{N}_{(2,1)}(A)$ so no link is created. The same procedure is repeated for $l = 3$ (nodes O and H are selected).

Note that even if the nodes are initially uniformly distributed throughout the space, over time changes in the node's attribute value distribution can result in a highly skewed distribution of nodes in the space. Our protocol benefits in some respects from a skewed distribution. If most nodes fall within a small portion of the space, this means that on average nodes will have fewer neighbors because many neighboring cells will be empty (e.g., A has no neighbor in $\mathcal{N}_{(2,1)}(A)$). However, as a previous example demonstrated query routing can be adversely affected by extremely large cells because queries do not use hierarchical routing within a cell. In Section 6, we describe our solution to eliminate large cells that can develop after initial deployment.

3.2. Query Routing

We illustrate query routing in RSS by example as depicted in Figure 2(b). Assume that node A is looking for $\sigma = 4$ nodes that have a network connection greater than 512 Kb/s and at least 4 GB of RAM. Graphically, this is represented by the dotted rectangle in Figure 2(b), representing the area $\mathcal{Q}$. Node A will first find that itself does not fall into $\mathcal{Q}$. A then increases its scope starting from the highest level neighboring cells, until it finds one that overlaps $\mathcal{Q}$. In our example, this process ends immediately with $l = 3$, since an overlap is found between $\mathcal{N}_{(3,0)}(A)$ and $\mathcal{Q}$. Hence, node A forwards the query to its neighbor $n_{(3,0)}(A)$ responsible for that subcell (node O in the example). The latter will proceed in the same way. However, to avoid backward messages, it considers only $\mathcal{N}_{(3,1)}(O)$ or lower-level cells.

Node O finds that $\mathcal{N}_{(3,1)}(O)$ partially overlaps $\mathcal{Q}$. It therefore forwards the query to T, i.e., $n_{(3,1)}(O)$. T first includes itself in the candidate set as it matches the query requirements. Then, since both $\mathcal{N}_{(3,0)}(T)$ and $\mathcal{N}_{(3,1)}(T)$ cannot be further considered to avoid backward propagation of the query, it can just consider $\mathcal{N}_{(l,k)}(T)$ with $l < 3$. It therefore routes the query towards $n_{(2,0)}(T)$, namely U, which fulfills the query requirements. Since A asked for 4 nodes, U continues to disseminate the query to S, in $\mathcal{N}_{(1,1)}(U)$, which also matches. Now, S cannot propagate the query further and thus replies back to U. Also U, T and O do not have alternative paths so, following the return path, the query goes back to A. Node A, however, can forward the query to H, since also $\mathcal{N}_{(3,1)}(A)$ overlaps with $\mathcal{Q}$. Here the propagation occurs as above and in the end the query reaches node L, whose attributes also match the query. This algorithm can easily be extended to support rapidly-changing attributes, such as the available disk space of a node. Instead of representing this attribute as an extra dimension, one can route queries according to other requested attributes, and let resources check locally if they match the dynamic attribute as well. This is not feasible in delegation-based systems.

As shown in Figure 2(b), query propagation follows a depth-first tree rooted at the originating node. This ensures that no loops are created. However, this tree is created dynamically each time a new query is issued, exploiting the links of the overlay network and thus dependent on the properties of the overlay. Compared with traditional approaches, where a single tree is used, this solution is more efficient due to a better load distribution and much lower maintenance costs, especially in presence of churn. With the self-adaptation discussed later, maintenance of the cell boundaries is autonomous and efficiently distributes load even in a highly dynamic system.

The query stops when all matching nodes have been found or the upper bound σ on the number of requested nodes has been reached. We refer the interested reader to [Costa et al. 2009] for a formal description of the routing protocol.

Table I. Default simulation parameters.

Parameter	Default value
Network size (N)	100,000 (PeerSim) 1,000 (DAS)
Query selectivity (f)	0.125
Max. no. requested nodes (σ)	50
Dimensions (d)	4
Nesting depth ($max(l)$)	3

3.3. Evaluation

To assess the performance of our routing protocol with fixed cell boundaries, we built two implementations. We deployed the first implementation on the DAS-3 cluster at VU University Amsterdam [DAS-3]. We emulated a system with 1,000 nodes by running 20 processes per node on 50 nodes. The second implementation runs on top of the PeerSim discrete event simulator [Montresor and Jelasity 2009]. This allows us to explore setups with up to 100,000 nodes.

Based on these setups, we evaluated the performance of our system in terms of efficiency and correctness. Efficiency is measured in terms of *routing overhead*, defined as the average number of hops travelled by a query through nodes that did *not* match the query themselves. Correctness means that each node that matches a query must be hit exactly once. We note that we always obtained 100% delivery (i.e., all matching nodes receive the query message) in all experiments where the system does not experience churn or changes in the node attributes. We discuss the effects of churn and dynamic changes on delivery in Section 4.3 and Section 5.3.

In all experiments, including the ones on the DAS, we first randomly populate the space with nodes following a *uniform* distribution and give them sufficient time to build their routing tables. Effectively, this allows us to consider the space as nicely built up from equally-sized d-dimensional cells. In later experiments, we drop the uniform distribution of nodes and consider a skewed one. In these experiments we fix the cell boundaries to evaluate the base protocol without self-adaptation.

We generate queries by selecting a subspace in the d-dimensional space such that it approximately contains a desired fraction f of the total number of nodes N, which we refer to as the *query selectivity*. Each query will therefore be satisfied by approximately $f \times N$ nodes. Different queries refer to different subspaces. Each query is then issued repeatedly from every node in the system. Unless otherwise specified, simulations are based on the default parameters depicted in Table I.

Hereafter we focus on the performance of the routing protocol and basic overlay and defer discussion of maintenance of the overlay to Section 4.

3.3.1. Effect of Network Size. Figure 3 plots the routing overhead of our system for different network sizes N. In all configurations, the overhead remains very small, on average below three messages per query. The overhead increases approximately logarithmically until 10,000 nodes, then decreases for large network sizes. This is due to the threshold $\sigma = 50$: when the network is densely populated, a query often reaches its requested threshold very early and does not need to iterate through all cells that may overlap with the query.

3.3.2. Effect of Query Selectivity. We now study the cost of queries with different selectivity, that is queries that match different fractions of the total system nodes. We studied two workloads. In the "best-case" scenario, each query is built such that it is satisfied by the nodes in a single cell and matches exactly the required number of nodes. The "worst-case" scenario consists of queries that require nodes from multiple subcells such that every dimension and cell level is represented. This represents the worst-case scenario because this

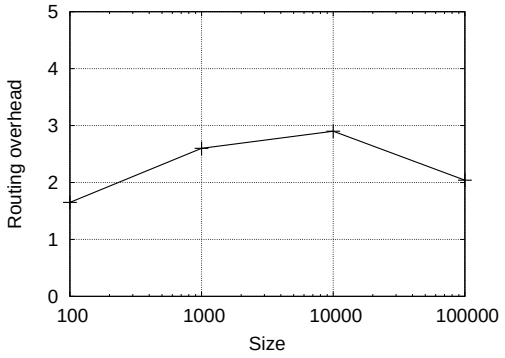
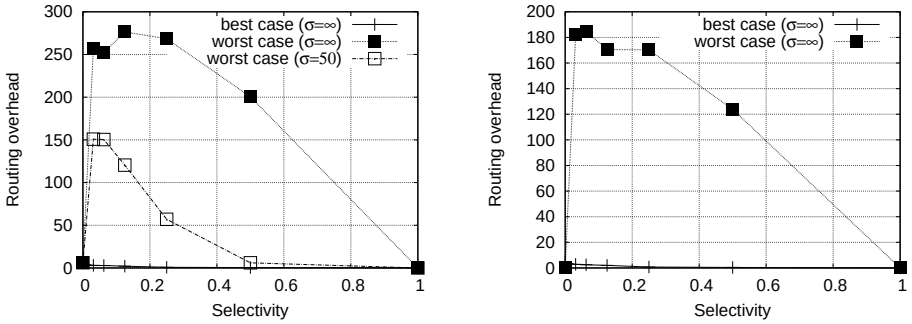


Fig. 3. Routing overhead vs. network size (PeerSim).



(a) PeerSim (b) DAS

Fig. 4. Routing overhead vs. selectivity.

requires to route the query on every dimension and level, thus increasing the path to reach all matching nodes.

Figure 4 shows results based on PeerSim and DAS. In the best-case scenario, the overhead remains negligible for all selectivity values. The worst-case scenario, however, shows higher overhead values, albeit still reasonable: e.g., in Figure 4(a), for $f = 0.125$ the overhead is 257 messages, to be compared with 12,500 matching nodes. This is due to the fact that queries that span multiple subcells must be split to cover all requested cells. This overhead decreases for queries with high selectivity: in these cases, the system contains less nodes that do not match the query.

In most cases, we can assume that a user wants to identify a limited number of nodes out of a large population of candidates that match the query. Due to the depth-first search of our algorithm, such queries can be stopped when they reach the threshold σ. This explains why experiments with $\sigma = 50$ always exhibit very low query overheads.

The overhead in the worst case does not change significantly between 100,000 (Figure 4(a)) and 1,000 nodes (Figure 4(b)). The reason is that the number of nodes to contact to reach the matching ones does not depend on the size of the network but on the topological properties of the space (i.e., the number of dimensions and the nesting depth), which are the same in both systems.

In our experience, the (non-)alignment of queries with the cell boundaries is one of the main factors which determines the performance of RSS. Ideally, cell boundaries would be selected such that most queries cover exactly one cell. This is however very difficult to impose, as users can in principle send any query workload to the system. We return to this

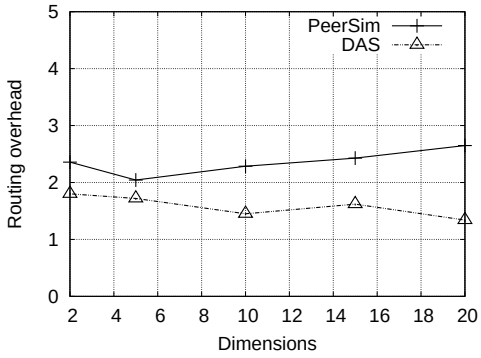


Fig. 5. Routing overhead vs. dimensions.

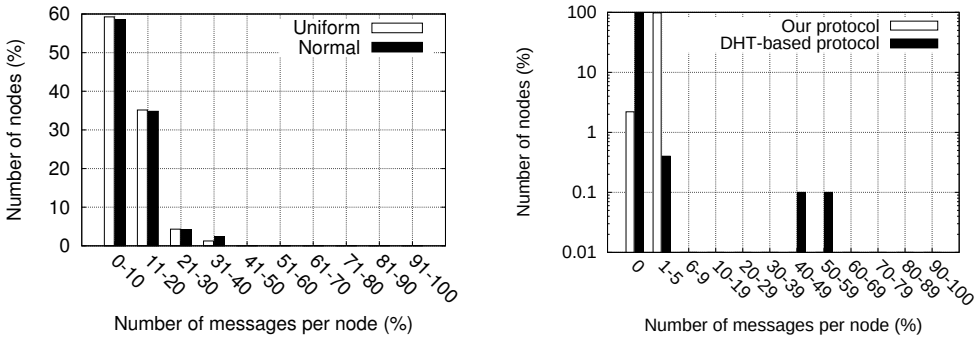
issue in Section 6 where we discuss self-adaptation algorithms to dynamically control the cell boundaries and minimize query cost.

3.3.3. Effect of the Number of Dimensions. A major difficulty in multidimensional peer-to-peer systems is to be able to handle a large number of dimensions, which in our system correspond to node attributes. Figure 5 charts the performance when using different numbers of dimensions, in both PeerSim and DAS setups. In PeerSim, the overhead increases slightly with the number of dimensions, while in the DAS it remains roughly constant. These variations, however, remain difficult to interpret, as such low overhead values typically fall within statistical error margins. Note that in all cases the overhead remains very low.

3.3.4. Load Distribution. In a large-scale system, it is important that the load imposed by the protocol is evenly distributed among nodes. Figure 6(a) shows the load in terms of messages (queries and replies) dispatched by each node. We exercised PeerSim with two different node distributions across the space. In the first one, each parameter of each node is selected randomly in the interval $[0, 80]$ using a uniformly random distribution. The second configuration creates a hotspot around coordinate $(60, 60, \ldots, 60)$. Nodes were distributed around that coordinate, with a standard deviation of 10.

In both cases, we observe that no node receives a load significantly higher than the others. This is due to the way neighbor lists are constructed (see Section 4). Even in dense areas of the hyperspace, each node selects its neighbors independently. The inherent randomness of this neighbor-selecting protocol evenly distributes the links across all nodes of a given cell which, in turn, leads to an even distribution of load among those nodes.

Figure 6(b) shows the load (as number of queries processed) seen by nodes comparing a DHT-based implementation to our approach in the DAS setup using 16 dimensions. Node attributes for each dimension are taken from the XtremLab BOINC project traces [XtremLab Project] that record node properties seen for more than 10,000 hosts in BOINC projects and are highly skewed. We use the Bamboo DHT [Rhea et al. 2004] and, as in SWORD [Albrecht et al. 2008], store a record of the nodes' attributes in the DHT at a key for each attribute value for each dimension. Searches are performed using a range query (implemented as an iterated search) across a random dimension until the requested number of nodes is found matching the query or the range is exhausted. Note that delegation produces a distribution with a heavy tail so that a few nodes receive a large number of queries in the DHT approach while our approach sends relatively few queries to all nodes, thus achieving an effective load-balancing. We chose SWORD for our comparison because it has been successfully adopted as resource selection service in PlanetLab and it is based on the publicly available Bamboo DHT. Nevertheless, the conclusions drawn can be extended to other DHT-based approaches as well.



(a) Uniform vs. normal distribution (PeerSim) (b) RSS vs. DHT-based approach (DAS)

Fig. 6. Node load distribution

3.3.5. Number of Neighbors per node. The next evaluation concerns the number of links that each node must maintain. Links belong to two categories. First, a node must maintain its *neighborZero* list, which links to at most every other node present in the same lowest-level cell. The number of cells in the system is $\left(2^d\right)^{\max(l)}$, where d is the number of dimensions and $\max(l)$ is the nesting depth. The cell number grows extremely fast with d and $\max(l)$, so we expect that in practice a lowest-level cell will contain only nodes strictly identical to each other (e.g., nodes belonging to the same cluster). However, even if that is not the case, we can relax this condition by demanding that the nodes in the same lowest-level cell are connected in an overlay. Such overlays are easy to construct and maintain [Jelasity et al. 2007].

Second, every node must maintain one link to a node in every neighbor cell for each dimension and level. Each node thus has $d \times \max(l)$ neighbor cells. However, because of the huge number of cells, even a 100,000-node system such as our PeerSim example will leave most cells empty. Nodes do not need to maintain a link to empty cells, so the actual number of neighbor links per node will be much lower than $d \times \max(l)$. This is confirmed in Figure 7(a): except for very low numbers of dimensions, the number of links per node, both in its *neighborZero* list and in its neighbor cells, is virtually constant. Similar results (omitted here for brevity reasons) are also obtained when varying $\max(l)$.

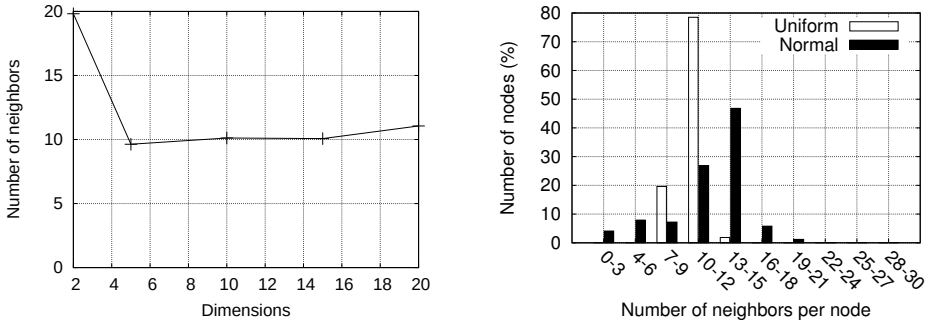
Figure 7(b) plots the distribution of the links per node in PeerSim, under uniform and normal distribution. In both cases, this number remains under 20 links in total. We note, however, that the normal distribution case requires slightly more links per node. This is due to the fact that *neighborZero* lists will grow in the cells around the hotspot.

4. OVERLAY MAINTENANCE

An important issue in the RSS protocol is to efficiently maintain the overlay in the presence of dynamic changes in the system. Nodes must be expected to join or leave the network frequently, for example due to failures and recoveries. Also, a node's attribute values may change during the system's lifetime. The distribution of requested values in queries will change significantly, possibly even on short time scales. In this section we show how the RSS overlay is proactively maintained in order to preserve routing correctness despite such changes. In this section we do not address yet self-adaptation for performance optimization.

4.1. Principles

Overlay maintenance is realized using previous work from our group in which nodes can dynamically self-organize into any pre-defined structure. The approach relies on a layered gossip-based protocol [Voulgaris and van Steen 2005]. In this organization, each node takes



(a) Neighbors vs. dimensions (b) Neighbors distribution

Fig. 7. Number of neighbors (PeerSim)

part in several overlays. Each overlay proactively maintains a list of links to other nodes in the system according to a well-defined criterion. In the simplest setup we need two layers to build and maintain the RSS overlay over time.

The bottom layer executes the CYCLON protocol [Jelasity et al. 2007], which aims to connect all nodes into a randomly structured overlay. Each node maintains a small list of K_c random links to other nodes in the system (with $K_c \ll N$). Each node periodically selects one neighbor randomly among K_c and exchanges a few of its links with those from its neighbor's list. This way, all nodes are periodically provided with a refreshed set of links to other randomly chosen nodes. The resulting overlay closely resembles a random graph in which failing nodes are quickly replaced and removed from the lists of other nodes. Such overlays have been shown to be extremely robust against partitioning even in the presence of churn and massive node failures.

The upper layer, named VICINITY, aims to maintain links to semantically related nodes rather than to random nodes. VICINITY executes a protocol very similar to CYCLON in that each node has a set of K_v links to other nodes and periodically exchanges information about a subset of its links K_c and K_v with its neighbors. However, unlike in CYCLON, nodes do not select links to keep in their sets randomly but choose them according to a preference function. All VICINITY links are associated with the attribute values of the nodes they represent. The preference function is constructed in such a way that the node that runs VICINITY selects at least one neighbor in each of its neighboring cells (assuming enough neighbors are available). This way, RSS nodes build an overlay required to route queries using the algorithm described in Section 3.

The two gossip layers interact in the sense that the underlying CYCLON layer continuously feeds the top VICINITY layers with random nodes to make sure the system remains connected. While the VICINITY layer are optimized for searching, the bottom CYCLON protocol is designed to handle network dynamics and to provide up-to-date information that will allow proper adjustments in the semantic lists.

These gossip protocols involve a continuous succession of so-called *g*ossip cycles, where at the beginning of each cycle the node contacts one of its neighbors to exchange information with. Typically, the length of a gossip cycle is in the order of seconds; for our experiments we chose a gossip cycle length of 5 seconds.

In order to join the system, a new node needs to know the addresses of a few other nodes (at least one) that already participate in the overlays, and with whom it will start gossiping. No particular bootstrapping action is necessary beyond this point. The new node will automatically be fed with the information it needs thanks to the regular gossiping protocol.

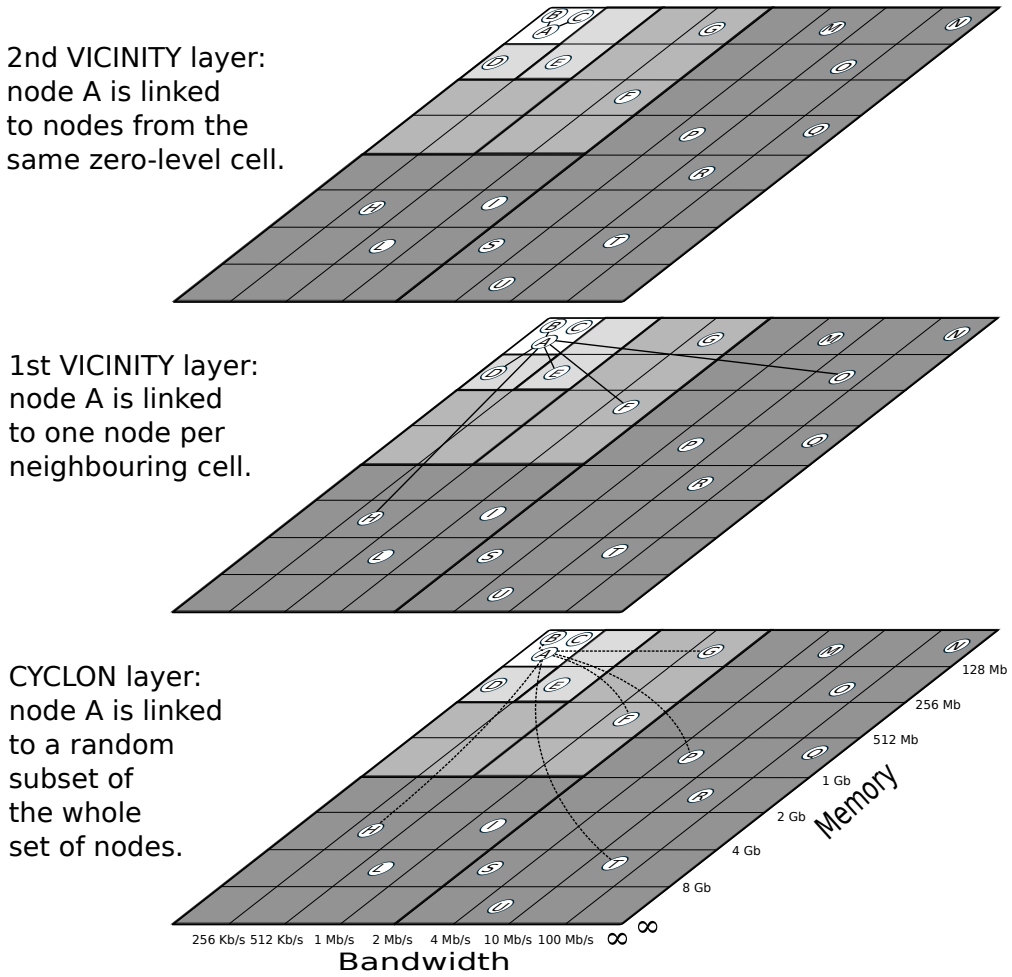


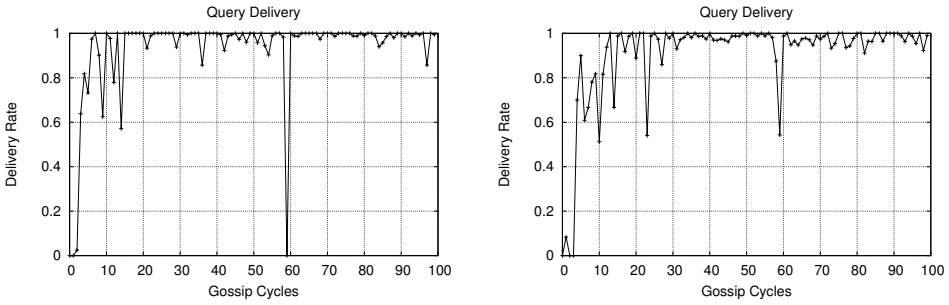
Fig. 8. Example of links maintained by a node in different gossip layers.

As discussed in [Voulgaris and van Steen 2005] and Section 4.3, this approach for self-organization converges very fast in the presence of major changes in node membership due to the fact that if two nodes are neighbors of each other, then there is a high probability that they have other neighbors in common.

4.2. Overlay maintenance implementation

In RSS we use two separate VICINITY layers on top of CYCLON. The role of the first layer is to maintain links to nodes in neighboring cells. Specifically, each node X maintains, for each level l and each dimension k, at least one link to a node placed in the neighboring cell $\mathcal{N}_{(l,k)}(X)$. When choosing their neighbors for this layer, the nodes also use their CYCLON caches, which provide fresh lists of random overlay members.

The second VICINITY layer aims to maintain links to other nodes located in the same zero-level cell. Unlike the first VICINITY layer which needs to find only one node in each neighboring cell, here the overlay aims to discover as many zero-level neighbors as possible. This dense inter-cell node mesh allows spreading queries efficiently within a zero-level cell in order to discover matching nodes.



(a) Churn = 0.1% per gossip cycle. (b) Churn = 0.2% per gossip cycle.

Fig. 9. Delivery vs. churn (PeerSim).

Figure 8 shows an example of the links maintained by a node in the three gossip layers. Any node can simply lookup in its two VICINITY neighbor lists to find nodes in specific cells, and thereby route queries.

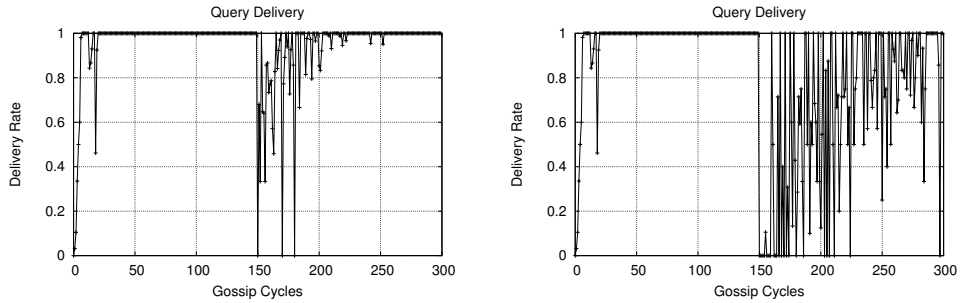
4.3. Evaluation

Experiments presented so far assume that the list of nodes remains stable. This is clearly unrealistic: any large network will exhibit a degree of dynamicity due to node joins and leaves. In particular, ungraceful node departures may represent an issue, since the routing tables of other nodes need to be updated to maintain correct routing. We claim, however, that no particular measure should be taken to handle churn. Instead, the underlying gossip-based protocol maintains correct routing tables continuously.

To support this claim, we evaluate our system with PeerSim in the cases of *churn* (a small percentage of nodes continuously joining or leaving the system) and *massive failure* (a large number of nodes failing simultaneously). We use as a metric the *query delivery*, which is defined as the fraction of nodes that are delivered in response to a query, compared to the total number of nodes that actually match the query. We also repeated the experiments on the DAS, which shows results comparable to those obtained on PeerSim. We omitted the charts for space reasons but they can be found in our previous paper[Costa et al. 2009].

4.3.1. Delivery under Churn. To evaluate the system under churn, we consider churn rates of 0.1% and 0.2% per gossip cycle. This means that respectively 0.1% and 0.2% of the nodes leave the system and re-enter it under a different identity every gossip cycle. Considering the gossip cycle length of 5 seconds, the 0.1% value corresponds to a churn rate of 0.2% per 10 seconds, which was observed in Gnutella [Saroiu et al. 2003]. However, many real-world systems are considerably more stable [Iosup et al. 2007]. We use a network size $N = 10,000$.

We measure the delivery over time by issuing one query every 5 seconds. Although in these experiments the queries are issued with the same frequency as the gossip cycles, there is no synchronization in the simulator between the query routing messages and the gossip messages. In order to assess the effect of churn with the highest accuracy, we did not set any threshold value for the queries, i.e., $\sigma = \infty$. Therefore, a delivery of 1 means that we reached all the matching nodes. As shown in Figure 9, the delivery rate remains high both for 0.1% and 0.2% churn rates. A delivery of 0.8 means that we retrieve 80% of all matching nodes. However, we expect most users of a real system to issue queries with a threshold. In such cases churn would only slightly reduce the number of reachable matching nodes to choose from, but most queries would be satisfied according to their specification. For instance, with a network size $N = 10,000$ and a selectivity $f = 0.125$, a delivery of 0.8 yields around 1,000 nodes (i.e., $N \cdot f \cdot 0.8$), which will often be above the expected number of nodes needed for a job.



(a) Failure = 50% (PeerSim). (b) Failure = 90% (PeerSim).

Fig. 10. Delivery vs. massive failure.

Also note that in these experiments, if a query cannot be propagated due to a broken link, the message is dropped. An alternative is to delay the query until the overlay has been restored by the underlying gossip protocols. This would have allowed delivery close to 1. However, this approach would increase latency, because nodes, upon the detection of a failed neighbor, would wait for the overlay to be repaired before forwarding the query.

4.3.2. Delivery under Massive Failure. This experiment studies delivery when a massive failure of a large fraction of the system happens simultaneously. We measure the delivery over time before and after the failure. Again, we do not use any threshold values or the previously discussed mechanism to avoid evaluation bias.

Figure 10(a) and 10(b) shows delivery in PeerSim when we remove respectively 50% and 90% of random nodes from the network at once. We submit a query to the system once in every 5 seconds (which is, once per gossip cycle). When the failure occurs many routing paths get disrupted, so the delivery oscillates across a broad spectrum. However, the system re-organizes itself rapidly. In the case of 50% simultaneous node failures, the system needs less than 10 minutes to recover completely. Only in the case of 90% simultaneous failures, the delivery could not be restored. The overlay gets partitioned by the massive failure so full recovery is impossible. A fallback protocol could be used in this case to bootstrap the system again.

5. ONLINE RECONFIGURATION

In a long-lived system, administrators will face the need to modify the set of attributes that describe the computing resources. For example, if the users of a new application need a specific library, the resource selection service must distinguish between the nodes on which the library is available and the ones on which it is not; furthermore, it might also be necessary to distinguish between different versions of the library. In this case, the administrators need to add a new attribute corresponding to the library. Conversely, when a certain library or feature is no longer used as a search criterion, the corresponding attribute should be removed.

Changes to the attribute search set must be performed online because the resource selection service is integral to using the system. Consequently, we designed a protocol that allows RSS to update its attribute set at runtime. In essence, this translates into modifying the dimensions of the RSS's attribute space while simultaneously handling search queries. In this section we introduce the update protocol and present a PeerSim-based evaluation.

In RSS, we consider three types of dimension set updates:

— **Adding a dimension:** this requires the administrator to specify the name of the new attribute and provide a code module that can calculate the value for the new attribute at any node in the system. Providing the appropriate security and access control mechanisms

```
UpdateDimensionSet(node):
neighbor ← select random neighbor
send (GossipRequest: my_timestamp) to neighbor
receive (GossipResponse: neighbor_timestamp) from neighbor
if my_timestamp < neighbor_timestamp then
   receive(configuration) from neighbor
   dimensions_list ← determine set of dimensions that need updates
   send (DimensionsRequest: dimensions_list) to neighbor
   receive (DimensionsResponse: updated_dimensions) from neighbor
   update dimension set
if my_timestamp > neighbor_timestamp then
   send(dimension_set_signature) to neighbor
   receive (DimensionsRequest: dimensions_list) from neighbor
   send (DimensionsResponse: updated_dimensions) to neighbor
```

Fig. 11. Pseudocode for a gossip exchange during which a node updates its dimension set.

is outside the scope of the paper. However, we envisage that every update (especially code) is signed by an administrator and can be securely verified by every node in the system.

— **Removing a dimension:** for this operation, the administrator has to specify the name of the dimension to be removed;

— **Updating a dimension:** this refers to changing the method by which an attribute value is computed (for example, when a more accurate or efficient CPU usage method is developed); the administrator has to provide new code that computes the attribute value.

The two main challenges in performing such dynamic updates are to quickly propagate the update to all system's nodes and to handle the user queries consistently while the update is in progress. The following two sections address these issues.

5.1. The update protocol

In RSS, new dimension sets are disseminated through a push-pull gossip protocol, which allows for rapid (exponential rate) propagation and has advantages of high scalability, robustness, and low overhead. Specifically, we use the same Cyclon protocol that stays at the base of the RSS overlay.

A dimension set update is introduced into the system by an administrator and can be submitted to any node in the system. The administrator associates every update with a timestamp. An update performed on a node creates a new node *configuration*, which consists of a timestamp and a list of dimension specifications. A dimension specification contains a dimension name, a code module for computing attribute values, and a timestamp of the last modification done to that dimension (or the time when the dimension was introduced if there have been no modifications).

We require that timestamps are totally ordered and newer configurations eventually overwrite an older configuration. We expect the frequency of updates be of the order of minutes, if not hours. This means that a loosely clock synchronization among nodes such as the one provided by the Network Time Protocol (NTP)[1] is sufficient.

In the RSS, each node periodically gossips with a randomly-selected Cyclon neighbor in order to check for dimension updates. We use a push-pull gossip model in which two gossiping nodes both send and receive their current configuration timestamps. If the timestamps are equal, nodes have the same configurations and no further communication is required. Otherwise, the node with the older configuration sends its dimension timestamps to the node with the newer configuration. The latter node then replies with the specifications for

[1] `http://www.ntp.org`

the dimensions that have been updated (i.e., those that have newer timestamps). The pseudocode for the gossip exchange is shown in Figure 11. As we show later in the evaluation section, this simple push-pull protocol propagates updates to virtually all nodes in a large system in just a few gossip cycles.

5.2. Query handling during updates

The RSS query routing protocol described in Section 3 requires that all nodes involved in query processing must use the same configuration (i.e., the same set of dimensions). Although the push-pull update protocol propagates new configurations very quickly to all nodes in the overlay, inevitably there are moments during reconfiguration in which multiple nodes have different dimension sets. Furthermore, since queries are processed in a certain non-negligible time, even if all nodes could switch their configurations instantaneously and synchronously, it might still happen that dimension sets are updated while queries are being processed.

In order to be able to handle queries correctly during configuration changes, RSS nodes temporarily cache old configurations. Each time a node receives a new configuration, it caches its current configuration together with the current set of neighbors. To prevent a maintenance overhead increase, the node runs the overlay maintenance protocols (CYCLON and VICINITY described in Section 4) only for the most recent neighbor set. Cached configurations and corresponding neighbor sets are stored read-only and are eventually discarded after a timeout.

When a query is generated by a node, it is associated with a configuration timestamp. All nodes that receive this query must make sure that they use the same (potentially cached) configuration to route the query consistently. In case a node receives a query for which it does not yet have a configuration, it can obtain the needed configuration from the query sender. However, as described later, we avoid such situations by delaying the use of new configurations.

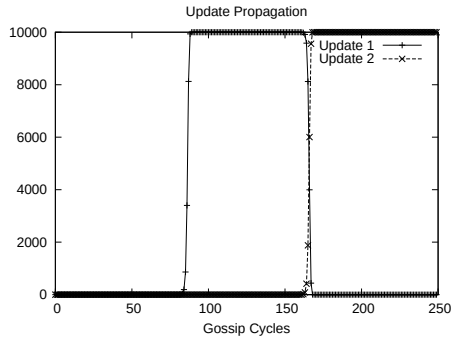
Another challenge in handling queries while dimension sets are being updated is related to neighbor discovery. When a node receives a new configuration, it starts building a new neighbor set based on the new dimension specifications. For a certain time the node has an incomplete neighbor set because the overlay maintenance protocols need to run for a few gossip cycles to discover all neighbors. In order to avoid routing queries over incomplete neighbor sets, RSS nodes delay the use of new configurations. Specifically, when a node generates a query and its current configuration is more recent than a certain threshold, the query is associated with an old (cached) configuration. Using this mechanisms, nodes make sure that queries are routed over fully converged and healthy overlays.

5.3. Evaluation

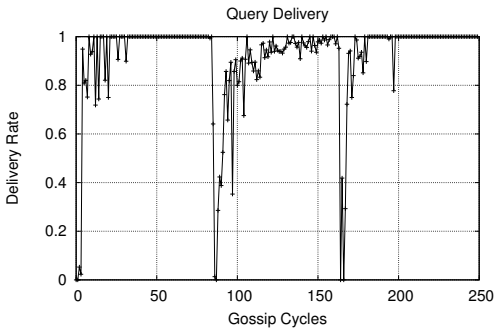
We evaluated the protocol for handling the reconfiguration of dimension sets in PeerSim. The goals of the evaluation were to assess how fast the new configurations are propagated in the system, and how much the query delivery rate is affected during such reconfiguration.

We present here the results of an experiment simulating a system with 10,000 nodes and with 5 initial dimensions, in which we perform two reconfiguration operations: adding a new dimension and then removing an older one. These operations are initiated, respectively, at 80 and 160 gossip cycles from the system startup.

This experiment shows that the updates are spread at an exponential rate among the nodes, which is consistent with the theoretical results proved by the existing literature on push-pull gossiping [Jelasity et al. 2007]. Figure 12(a) shows the results of our experiment in this respect. On the y axis we represent the total number of nodes that are currently using the updated dimension sets (each curve corresponding to one of the two updates). The x axis represents the time. As can be seen from the figure, it takes only approximately 5 gossip cycles (25 seconds) to propagate an update in a 10000 nodes system. The curves



(a) Propagation of the dimension set updates among nodes.



(b) Query delivery rate.

Fig. 12. System behavior during two reconfiguration operations.

are used to represent the latest update obtained by the nodes; thus, once the second update is issued, the first update will be used by fewer and fewer nodes.

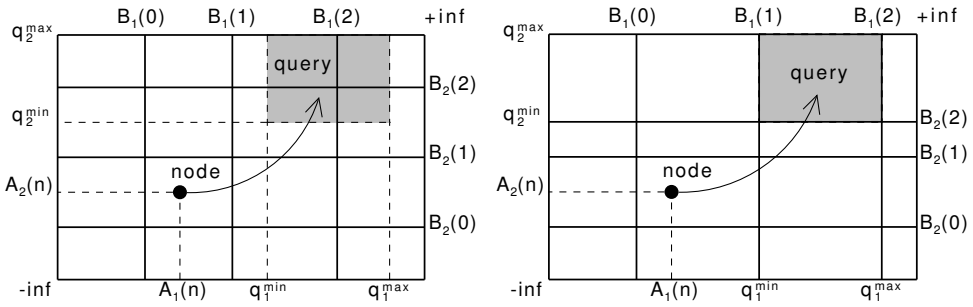
Figure 12(b) shows the query delivery rate for the same experiment. The queries are issued in this experiment at every 5 seconds. We noticed that, after a reconfiguration, the system needs approximately 20-30 gossip cycles (100 - 150 seconds) to completely rebuild the overlay and achieve a high delivery rate. In this experiment we again performed an exhaustive search in the system (i.e., asking for *all* the nodes that satisfy some given requirements), which is more than users normally require.

The communication overhead associated with a dimension set update is small. The mechanism used to update the dimension set is similar to the one employed to update the cell boundary sets for self-adaptation. We estimate this overhead in Section 6.6.5.

6. SELF-ADAPTATION

The cell boundaries in RSS can be chosen arbitrarily as the routing algorithm works correctly for any division of the virtual space. However, we observed that the placement of cell boundaries has a strong impact on RSS's performance. In this section, we describe the self-adaptive algorithm that continuously runs at all RSS nodes and makes autonomous decisions on cell boundary placement based on the current system load and node characteristics. As we show later, this self-adaptive algorithm reduces query routing overhead in RSS up to 4 times.

The self-adaptation mechanism is integral to a long-lived resource service that can survive significant changes in query load and the composition of the nodes in the system. This mechanism enables RSS to support occasional changes in attribute values. However, using it to support fast-changing attributes such as the CPU load or the current amount of available memory would incur very frequent reconfiguration of the RSS overlay and, hence, it would be highly inefficient. To avoid this overhead, RSS supports fast-changing attributes using local filters. Queries are routed only based on static (or moderately changing) parameters. Each matching node can then locally check if it also matches additional dynamic properties, and decide to add itself to the result or not based on this local information. This reduces the RSS reconfiguration overhead and allows queries to always use up-to-date information about dynamic parameters. A similar approach of performing the system-wide search based on static parameters, and then checking the dynamic values, was recently used by [Sharma et al. 2011].



(a) Suboptimal cell boundaries. (b) Optimal cell boundaries.

Fig. 13. RSS query routing in two dimensions. Every node is placed in a multidimensional space according to its attributes, and queries are represented by hyper-rectangles of the attribute space. Optimum boundary placement for a given query forms a single cell coincident with the query.

6.1. Performance Model

As discussed in Section 3, the cost of a query routing from a node where a query is submitted to the cell matching this query, in terms of message transmissions, is bounded by $O(D \log K)$. Since all messages are sent sequentially, this cost also determines query latency. Further, the cost of searching within a zero-level cell (depth-first search) is proportional to the number of nodes that belong to that cell. Thus, large zero-level cells might generate a high overhead as they potentially require sending messages to many nodes that do not match the query. The boundaries between cells should therefore be defined such that the nodes be more or less evenly distributed among cells.

User queries represent another factor that should be taken into account when defining the cell boundaries. Let us consider an example shown in Figure 13(a). A sample query overlaps with 4 cells in a 2-dimensional space. The query needs to be routed up to 4 different cells and a number of non-matching nodes potentially needs to be visited in each of these cells. For this query, a better boundary placement would have been the one represented in Figure 13(b), where only one cell needs to be explored and no non-matching nodes have to be traversed. As we have shown in Section 3.3.2, there is a significant difference in overhead between best-case queries (those that fit into one cell in the multi-dimensional space) and worst-case queries (those that span a large number of cells). Hence, as a general heuristic, it is desirable to have cells that fully overlap with frequently specified user queries.

For the system administrator it would be difficult to guess the optimal RSS boundary settings: firstly, because of the large number of nodes, and secondly because of the system's dynamism. The distribution of node attribute values changes in time due to churn (thus changing the size of the cells), and so do the queries issued by users. Of course, tuning the cell boundaries to adapt to each received query would be impractical; however, making periodic boundary adjustments based on statistical information for the recently received queries can bring important improvements. Thus, the goal of our adaptation algorithm is to periodically re-configure cell boundaries, according to the heuristics introduced above, in order to minimize the query handling overhead and distribution skew.

6.2. Decentralized Self-Adaptation Approach

It should be emphasized that although RSS is fully decentralized with respect to node membership and query management, the set of cell boundaries in the attribute space represents a global configuration parameter on which all the nodes must agree. If the nodes have different views on the cell boundaries, several query routing errors may occur such as queries not being routed to some of the matching nodes or even routing loops. Our self-adaptation

algorithm must guarantee that all nodes eventually agree on how the attribute space is divided into multi-dimensional cells.

Addressing such a coordinated self-adaptation in a distributed system requires finding a sweet spot between a fully decentralized design where each node may choose its own settings autonomously and a centralized one in which a single leader chooses a global configuration (such as an attribute space decomposition) and imposes it on the rest of the system. The first design faces the risk that nodes choose inconsistent configurations, while the centralized approach introduces a single point of failure and a potential performance bottleneck.

In order to perform efficient coordination, RSS relies on self-elected leaders, where each leader is in charge of carrying out one instance of the adaptation protocol through the whole system. Instances of the protocol are run periodically, with the purpose of re-computing the optimal set of cell boundaries according to the current attribute values distribution and query workload. At each round, each node may elect itself as a leader with a probability inversely proportional to the number of nodes in the system. This allows maintaining the average frequency at which leaders emerge from the system. However, it does not prevent multiple adaptation instances from executing simultaneously in the system. We therefore impose a global order between protocol instances using a totally ordered timestamp so that newer instances can supersede older ones. This also allows nodes to garbage collect old instances whose leader failed before the completion of its task.

To obtain a global order among the protocol instances, we use a combination of physical timestamps and unique node identifiers. For the physical timestamps, we assume, as discussed in Section 5.1 that the nodes use a clock synchronization protocol like NTP. The synchronization errors introduced by such protocols are at least two orders of magnitude smaller than the typical gossip frequency.

6.3. Gathering Monitoring Information

An adaptation leader needs information about the global system state in order to select new cell boundaries for the RSS attribute space. Here as well, we observe a necessary tradeoff regarding the quantity of information made available to the leader. On the one hand, bringing detailed information about each node may allow the leader to make accurate complex choices, but the costs of gathering such exhaustive information may be prohibitive, especially in large-scale distributed systems. On the other hand, distributed aggregation algorithms can efficiently compute functions such as the average of some attribute value across the system. However, a single aggregate value such as an average might not provide enough detail to optimize the global configuration. In particular, average values are very sensitive to the presence of a small number of outliers in the system.

We argue that a reasonable tradeoff consists of estimating the statistical distribution of node attributes across the system. A statistical distribution captures essential information about the system: it shows the full spectrum of node characteristics, and the proportion in which they exist. This allows nodes to make complex decisions where a balance between multiple contradictory requirements is often involved. In particular, in RSS we are interested in the statistical distributions for node attribute values and also for query ranges.

Estimating the statistical distribution of an attribute across a large-scale distributed system can be realized both accurately and inexpensively. In this paper we use our own Adam2 algorithm which efficiently approximates node attribute distributions in a fully decentralized manner [Sacha et al. 2010]. Adam2 approximates cumulative distribution functions by estimating their values in a few carefully selected points and interpolating between these known points. Each distribution approximation is produced by a sequence of aggregation instances composed of a fixed number of gossip rounds. Instances iteratively refine the interpolation point placement. Adam2 is also able to tune its own approximation accuracy during the refinement process.

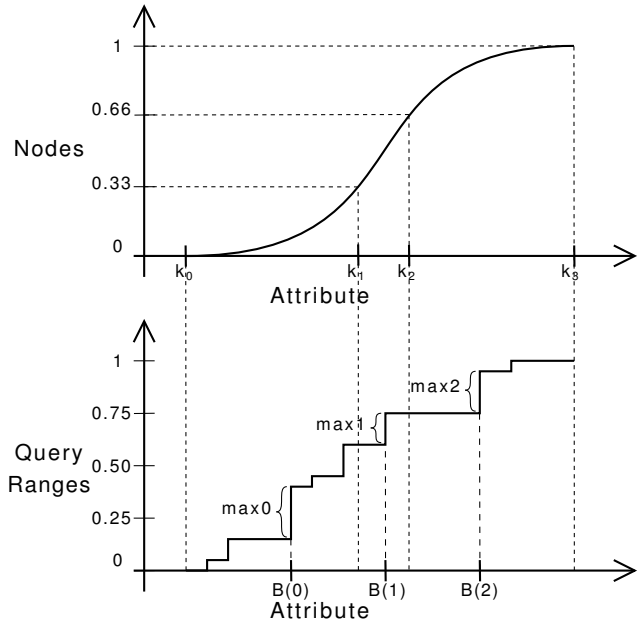


Fig. 14. The boundary calculation algorithm first finds $(k_0, \ldots, k_3)$ to balance the number of nodes in each partition (above), and then places a boundary B in each partition at the largest grouping of query endpoints (below).

In the self-adaptive RSS, instances of Adam2 are continuously initiated by self-elected leaders. A leader maintains two distribution approximations for each dimension d: the node attribute distribution $Attr_d$, and the query range distribution $Query_d$. The attribute distribution is a function $Attr_d : \mathbb{R} \to \mathbb{R}$ defined such that $Attr_d(x)$ is equal to the fraction of nodes in the system that have a value for attribute A_d below x. This distribution provides the leader enough information to place cell boundaries to balance the number of nodes in each cell.

Monitoring the distribution of queries is harder than monitoring node attributes because queries are composed of ranges in each dimension. As we discuss next, for the placement of cell boundaries, the leader is interested only in the distribution of the endpoints of query ranges. We thus define the query distribution for dimension d as a function $Query_d : \mathbb{R} \to \mathbb{R}$ such that $Query_d(x)$ is equal to the fraction of all the endpoints of query ranges (upper or lower) for dimension d. In order to reduce the influence of old queries and to reduce bookkeeping, nodes cache received queries for only T time units to form the query distribution.

6.4. Computing New Configurations

Using the distributions of node attributes and query ranges obtained from Adam2, the leader of the instance creates a configuration consisting of a new unique, totally-ordered timestamp and (possibly new) cell boundaries. The new configuration is then installed by the leader and will then be spread by gossip as discussed shortly. Note that the leader installs a new configuration even if it differs only slightly from the previous configuration because the precise placement of cell boundaries is important to reduce query overhead.

To create a configuration with good expected performance, the leader must solve an online optimization problem with future queries assumed to be similar to recent past queries. As discussed previously, the optimal placement of boundaries for a particular query creates a

cell precisely the same size as the query to minimize routing costs to only one cell and within the cell all nodes match the query. A larger cell than the query runs the risk of routing to some nodes within the cell that do not match the query, while smaller cells than the query result in higher routing overhead between all the cells that overlap the query. The tradeoffs in cell size are complex and interdependent.

We use two heuristics to simplify the calculation of cell boundaries: (i) we attempt to balance the number of nodes in each cell to prevent the formation of overly large or small cells; and (ii) we attempt to place the boundaries of the cells coincident with the endpoints of the most frequent query ranges. Specifically, the cell boundary algorithm calculates a new set of K cell boundaries $B_d(0), B_d(1), ...B_d(K-1)$ for each dimension d independently. The calculation follows our two heuristics one after the other, as shown in Figure 14. In the first step, the node attribute distribution is used to define initial intervals for boundary values such that the nodes are roughly balanced between all cells. Specifically, $K+1$ points are calculated, $k_0, k_1, ...k_K$, such that $Attr_d(k_i) = \frac{i}{K}$. The calculation of k_i points is straightforward since $Attr_d$ is a non-decreasing function approximated by line segments, which can be easily inverted. The final value for each $B_d(i)$ boundary is later chosen such that $k_i \leq B_d(i) < k_{i+1}$ so that the cell determined by any two consecutive boundaries $B_d(i)$ and $B_d(i+1)$ contains at most $\frac{2}{K}$ of all nodes.

In the second step, precise cell boundaries are calculated using the query distribution. The goal of this phase is to place cell boundaries specifically at the most frequent query range endpoints. The reason we chose the location of the most frequent endpoints instead of using more complex clustering of endpoints was not the simplicity of this solution. The benefits of reducing routing overhead only appear if the cell boundaries are exactly those of the query: A small overlap of the query with another cell can force the query to be routed to all nodes in the other cell. Hence, precise placement of cell boundaries is very important to reducing routing overhead in the RSS.

The most frequent range endpoints are easily identified by the point of the largest change in $Query_d$. The height of each change in the query distribution function is by definition equal to the frequency of the corresponding query endpoint. Boundary $B_d(i)$ is placed at the greatest change in the query distribution between points k_i and k_{i+1} as shown in the bottom of Figure 14. If $Query_d$ does not change between points k_i and k_{i+1}, boundary $B_d(i)$ is simply defined as $\frac{k_i+k_{i+1}}{2}$. By aligning cell boundaries with common query ranges we reduce both the hierarchical routing overhead since queries intersect with fewer cells, and the intra-cell routing overhead since fewer non-matching nodes have to be visited when exploring partially overlapping cells.

6.5. System Reconfiguration

Relying on self-elected leaders allows the RSS nodes to take unambiguous decisions on the attribute space decomposition. However, to achieve a consistent system state we must also ensure that all nodes receive the same new configuration and transition to the new configuration in a coordinated manner, without a significant service disruption.

In order to spread the new set of boundaries from the leader to the rest of the system, we extend the gossip protocol that we used in Section 5.1 for adding and removing dimensions. In the extended protocol, a node configuration contains not only dimension specifications but also current cell boundaries. Hence, when a self-elected leader decides on a new set of cell boundaries, it generates a new configuration, which is then quickly propagated to other nodes in the system by the push-pull gossip protocol.

Since the leaders are elected probabilistically, it is possible that multiple leaders choose different cell boundaries and start concurrent reconfigurations. However, the total order on timestamps guarantees that only one set of cell boundaries becomes the latest and eventually overwrites all other boundary sets.

6.6. Evaluation

The evaluation of our self-adaptation protocol aims to estimate the performance improvement that this protocol brings to RSS. We evaluate our protocol in PeerSim, by comparing two versions of RSS: one that uses the self-adaptation protocol, and one that does not use it.

We consider two test cases in our evaluation: an adaptation to changes in the node population, and an adaptation to changes in query workloads. In both test cases, we simulate two types of changes: sudden changes (e.g., an addition of a new computing cluster to the system, or a switch from one type of application to another), and gradual changes (e.g., a system in which old machines are gradually replaced by new machines, or a slow transition in the type of jobs that users tend to run in the system).

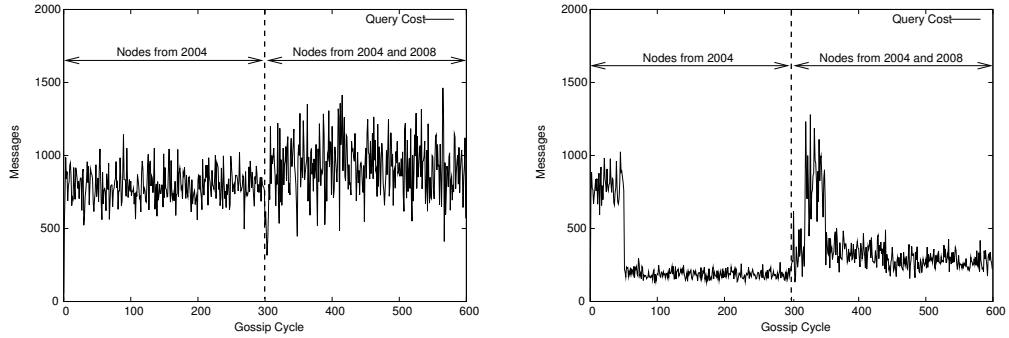
We assess the improvement in the RSS performance by measuring the RSS total query cost, defined as the average number of nodes traversed by a query. This metric captures both the query routing cost and query latency, since RSS queries traverse nodes sequentially. We also investigate the impact of our self-adaptation protocol on the RSS responsiveness by measuring the query delivery rate, defined as the average fraction of nodes correctly discovered by a query. Finally, we measure the extra maintenance cost introduced in the RSS by our self-adaptation protocol.

6.6.1. Experimental Setup. Although we evaluate our system through simulation, we use real-world data to initialize node attribute values and several types of queries that closely resemble the workloads from current Grid systems. Specifically, we obtained descriptions of over 300,000 machines that participated in the the BOINC volunteer computing project between 2004 and 2008 [Anderson and Reed 2009]. Based on these machine descriptions, we initialize the following four node attributes in the RSS: measured CPU performance in FLOPS, measured downstream bandwidth, amount of installed memory, and amount of installed disk space.

We exercise the system with several types of synthetic query workloads that have similar characteristics to the workloads observed in real Grid systems. Although a number of job traces from Grid systems are available [Iosup et al. 2008], we could not use them directly in our experiments because they mostly contain information about job runtime characteristics (e.g., total running time, amount of used memory) and give very little information about node characteristics required for job execution. In our experiments, we use the following three workload types:

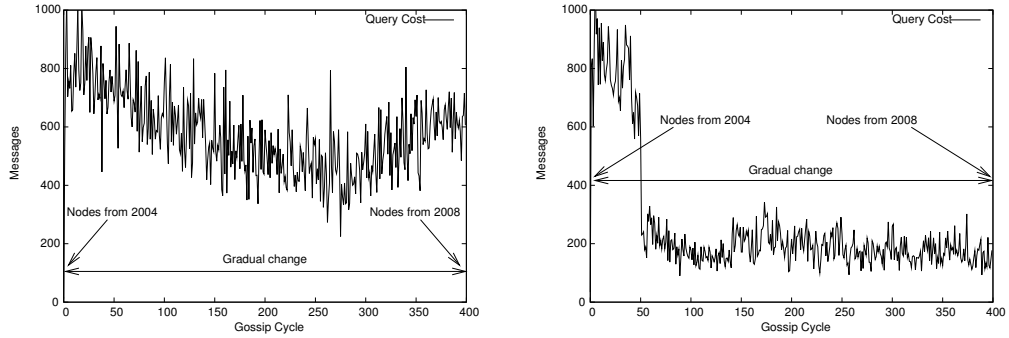
— **Bag-of-tasks:** a workload in which a few specific queries appear very frequently. This corresponds to the "bag-of-tasks" type of jobs, that contain a large number of very similar tasks (and thus, a large number of identical job submissions). We generate this workload by creating three queries that account for 75% of all submissions (25% each) and drawing all other queries randomly.
— **Coarse-grained:** a workload which simulates user-generated queries. In such queries, attribute ranges are specified in course-grained units. For example, the amount of RAM is specified in multiples of 512 MB. We generate these queries by rounding up random attribute intervals.
— **Random:** a workload in which all the queries specify random intervals for attribute values. We use this workload as a base for comparison with the other workloads.

6.6.2. Adaptation to Changes in Node Properties. The statistical distribution of node properties may change dramatically when new machines are added to the system, or when they replace older ones. To simulate such situations, we use two sets of node properties based on the BOINC traces from years 2004 and 2008. For this particular experiment we replaced one node attribute (available downstream bandwidth) with the installed kernel version: this attribute suffers much more changes across the years, and allows to stress our system better.



(a) Without self-adaptation (b) With self-adaptation

Fig. 15. Total query cost for a sudden change in the node properties (with random queries).



(a) Without self-adaptation (b) With self-adaptation

Fig. 16. Total query cost for gradual changes in the statistical distribution of node properties (with random queries).

The case when a new computing cluster is added to the system creates a sudden change in the statistical distribution of node properties. We simulated this case by starting RSS with 5,000 nodes with attribute values obtained from the 2004 traces. After 300 gossip cycles, we added 5,000 more nodes with attribute values from the 2008 trace. We used a workload composed of random queries for these experiments. The total query cost, with and without the self-adaptation protocol running, is shown in Figure 15.

The first part of Figures 15(a) and 15(b) show the effect of self-configuration in the RSS. Both systems start with the same set of query boundaries chosen by the human operator, and experience a query cost in the order of 800 messages per query. In the adaptive system, these costs drop by a factor 4 after the first system reconfiguration. At time 300, both systems see a cost increase. Part of this increase is due to the fact that the size of the system is doubled, and therefore the number of nodes matching the queries also roughly doubles. The adaptive system also sees an additional cost increase due to the fact that its configuration is suddenly ill-suited to the workload. It however quickly adapts to this new situation and returns to an average cost four times lower than the non-adaptive system.

The high cost values obtained in this experiment (and also in the other experiments presented in this section) are due to the fact that, to stress the system, we did not set any threshold for the number of nodes that should be returned by the queries. Thus, we perform exhaustive searching in the overlay, which incurs a much larger cost than the limited search requests which are usually issued by real-world applications.

Figure 16 presents similar simulation results for a situation in which the node properties gradually change from one distribution to another. We create this change by starting with 5,000 nodes from the 2004 BOINC trace, and subsequently replacing a few nodes at each gossip cycle with new ones drawn from the 2008 trace. Again, the adaptive system shows much better performance than the non-adaptive one. The non-adaptive system sees a relative performance improvement until cycle 250. This is explained by the fact that, in that phase of the experiment, there is a balance between the number of old and new machines, and the nodes are distributed more evenly into cells. The adaptive system, on the other hand, issues several relatively minor reconfigurations, and maintains a constant performance despite the workload variations.

6.6.3. Adaptation to Varying Query Workloads. We now evaluate RSS adaptation to variations in the query workloads it receives. We simulated 10,000 nodes, with attributes drawn from the 2008 BOINC trace. We first consider sudden workload changes by switching the query workload from one type to another every 300 gossip cycles. We start the experiment with random queries. We then switch to bags-of-tasks, then coarse-grained queries, and then bags-of-tasks again (similar to the first workload but with a different set of frequent queries).

Figure 17 shows the performance of the RSS in the adaptive and non-adaptive cases. The non-adaptive system observes no significant cost difference between workloads, except for the coarse-grained workload. This workload can in fact be considered as a best case for the manual configuration of the system, since the query ranges are aligned to the same values as the cell boundaries.

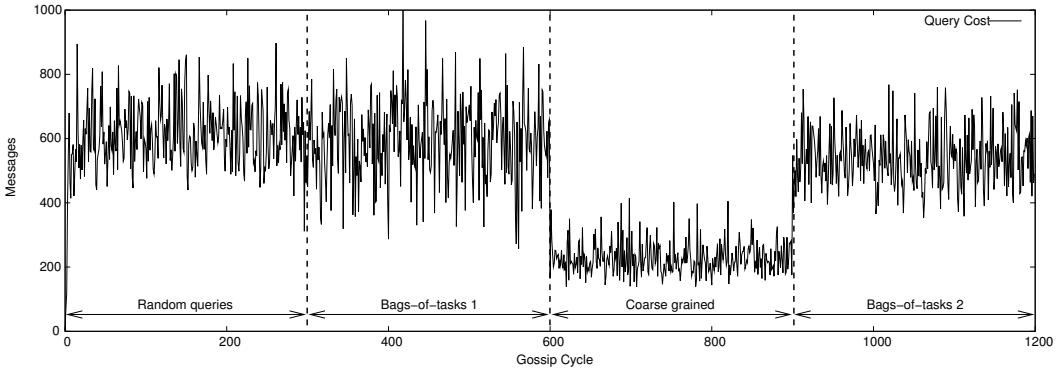
We can observe that here as well the self-adaptation protocol brings a significant cost improvement. The first decrease in the cost, obtained at a short time after the experiment's startup, is due to the system adapting to the distribution of node properties. When the workload changes at gossip cycle 600 and 900, we see a small cost increase due to the fact that the previous configuration does not work best with the new workload. However, the costs quickly decrease again thanks to self-adaptation. In particular, for the coarse-grained workload, we can see that the self-adaptation algorithm finds a configuration very close to the manually-configured "optimal" one from the non-adaptive system.

In order to evaluate the system's behavior for a (more realistic) gradual change of workload, we model a slow transition from the coarse-grained workload to a bag-of-tasks. Figure 18 shows the results of this experiment. In the first 100 gossip cycles, all the queries submitted to the system are coarse-grained. Then, we introduce bag-of-tasks queries with an increasing frequency besides the coarse-grained queries, until the last 100 gossip cycles when all the queries are bag-of-tasks. At the beginning of the experiment both systems use the same "optimal" set of boundaries so their performance is similar. When the workload starts to change, however, the non-adaptive system sees its costs increase twofold while the adaptive system efficiently controls reconfigurations and maintains a constant performance.

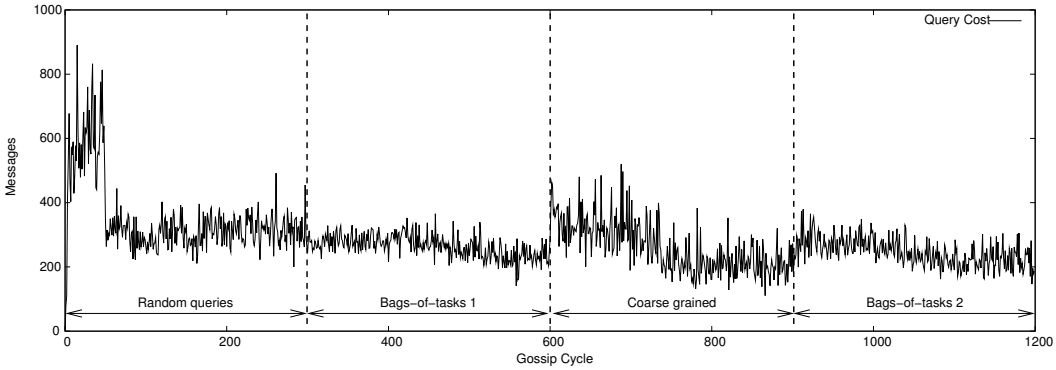
6.6.4. Impact on the Query Delivery. We now evaluate the impact of a runtime reconfiguration on the query delivery – that is, the number of nodes found by RSS divided by the total number of nodes that actually match the query.

When the system starts, it takes 100 to 200 gossip cycles for each node to build a full set of neighbors. In a system with no churn nor runtime reconfiguration, the query delivery converges to 100%. When a reconfiguration occurs, each node needs to rebuild a new list of neighbors according to the new cell boundaries. However, when the reconfiguration is small, most of the previous neighbors can be reused in the new list. Only very few neighbors need to be found anew.

Reconfigurations have a second type of impact on query delivery: once a query is submitted to the system the routing algorithm assumes that all nodes use a single consistent set of cell boundaries. When a node receives a query that refers to an old set of boundaries



(a) Without self-adaptation



(b) With self-adaptation

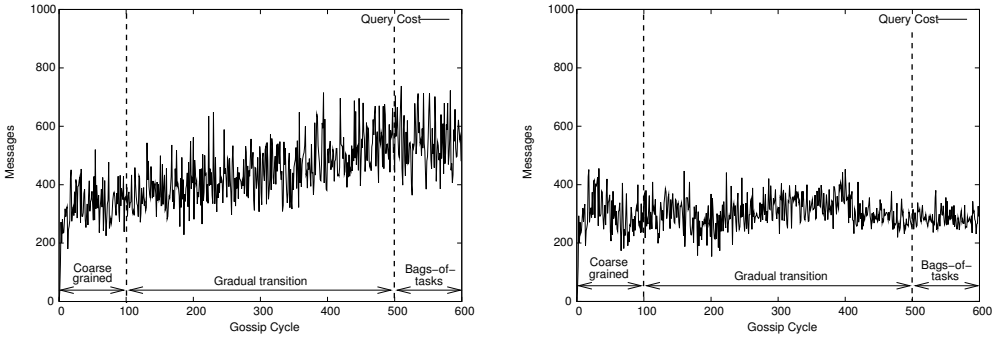
Fig. 17. Total query cost for sudden changes in query workloads (nodes from 2008 only).

that it does not maintain any more, all it can do is terminate the query, leading to poor query delivery.

Figure 19 shows the query delivery during the same experiment as in Figure 18: the workload gradually changes from coarse-grained queries to bags-of tasks. We show two cases: one in which each node immediately forgets its previous configuration when it receives a new one, and the case where nodes maintain a read-only cache of recent configurations. When previous configurations are not cached, the system experiences a large drop in query delivery at each adaptation. This is due to the fact that most queries present in the system at the time of reconfiguration will be terminated prematurely due to configuration inconsistencies. Figure 19(b) shows that this effect disappears when using the caching policy. In this case, delivery decreases only at the times of major reconfigurations when nodes need to seek for new neighbors. In all cases, even during reconfiguration, delivery remains high, which should remain sufficient for ensuring continuous service of the RSS within the computing grid.

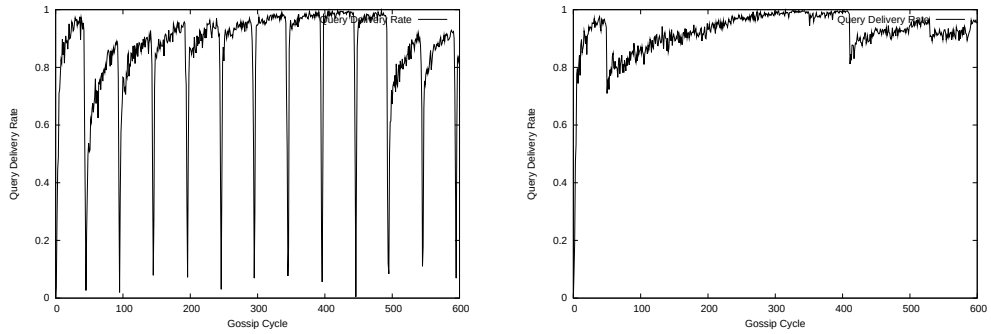
6.6.5. The Cost of Self-Adaptation and Overlay Maintenance. An important goal of the adaptation algorithm is to incur only a small cost overhead compared to the system that is optimizing. The most significant part of this overhead is the protocol's communication cost, which we estimate as follows.

The two main protocol phases that involve communication among nodes are the attribute CDF estimation through the Adam2 protocol and the dissemination of new boundary sets. As shown in [Sacha et al. 2010], the Adam2 protocol needs up to 100 gossip cycles to



(a) Without self-adaptation (b) With self-adaptation

Fig. 18. Total query cost for a gradual change in the query workload (nodes from 2008 only).



(a) (b)

Fig. 19. Query delivery rate, without (a) and with (b) caching older configurations.

generate an accurate distribution approximation. During this time, a node typically sends a total of 160 kB of data for each attribute and receives a similar amount of data. Considering a periodicity of one gossip round per 5 seconds, during the aggregation phase each node would need an average upstream bandwidth of 0.32 kB/s for each attribute. For an overlay with 4 attributes, as the one used in our tests, the needed bandwidth during aggregation is 2.56 kB/s for each node.

Besides the overhead brought by the self-adaptation protocol, there is also a communication cost associated with the overlay maintenance protocols: one CYCLON layer and two VICINITY layers. These protocols use gossiping to exchange the list of neighboring nodes. In our experiments, for each protocol layer a node sends and receives, in each gossip round, a list containing 15 other nodes. Each node is uniquely identified by the (IP address, port) pair, which requires 6 bytes per node. Therefore, the size of a message sent or received in a gossip round by a protocol layer is therefore smaller than 100 B, assuming 6 bytes (IP address and port) are used per node. Thus, the cumulative size of the messages sent or received by the overlay maintenance protocols is at most 300 B. Considering a periodicity of one gossip round per 5 seconds, each node would need an average upstream bandwidth of 60 B/s and a similar downstream bandwidth. Cumulating this with the bandwidth used by the aggregation protocol, we obtain a total used bandwidth of 2.68 kB/s for each node during aggregation.

The dissemination of new boundary sets has a significantly lower communication overhead. In order to decide whether it is necessary to reconfigure the boundary sets, the nodes

periodically exchange their current timestamps of the sets. This information can be added to the regular gossip messages used to maintain the overlay, increasing their size with only 4 B. When a new boundary set is issued, each node receives it only once; for one attribute, the size of the set is normally less than 150 B. Updating the overlay's dimension set (discussed in Section 5.1) has a similar overhead to the one associated with updating the boundary set.

Caching old configurations during boundary set or dimension set updates incurs a small memory overhead in the order of a few kilobytes.

Taking into account the time needed to propagate the new boundary sets after they are calculated, it takes a total of up to 200 gossip cycles to effectively reduce the routing overhead after a change in the system. Considering a gossip cycle of 5 seconds, the system will be properly reconfigured within 15-20 minutes. In case a faster reaction time is necessary, gossip cycles can be initiated more frequently, for example once per second; this would result in a reconfiguration time of 3-4 minutes.

7. RELATED WORK

In this section we discuss previous research works related to resource selection in large-scale distributed systems. We also introduce some relevant works that approach decentralized self-adaptation, which is essential in the RSS to maintain good performance despite variations of workload and node attributes.

7.1. Resource Selection

Traditional approaches for resource selection in distributed environments use centralized or hierarchical architectures in which a few servers keep track of all the resources in the network and offer lookup functionality to the users [Zanikolas and Sakellariou 2005]. While these solutions are well-suited for clusters or small collections of PCs, they exhibit scalability issues when the system size increases [Anderson and Roscoe 2006]. Hence, in recent years, researchers put forth a large effort to devise decentralized solutions addressing large-scale scenarios [Foster and Iamnitchi 2003].

The vast majority of these systems exploit DHTs to map resources to nodes in order to distribute the search operations across different nodes [Ranjan et al. 2008]. Early approaches maintain a separate DHT per attribute: a query is executed in parallel on every overlay network and results are then intersected [Spence and Harris 2003; Cai et al. 2003]. Alternative approaches reduce the d-dimensional space to a 1-dimensional space by means of a Space Filling Curve, thus reducing the problem to routing in 1-dimensional space [Ganesan et al. 2004; Schmidt and Parashar 2003]. More recent work, inspired by CAN [Ratnasamy et al. 2001], partitions the d-dimensional space into smaller blocks that are assigned to specific nodes, which are responsible for all resources falling in that block [Bharambe et al. 2004; Gupta et al. 2004; Schütt et al. 2007; Tanin et al. 2007]. Finally, SWORD [Albrecht et al. 2008] explores several centralized and decentralized solutions for resource discovery; in the decentralized solution, a different DHT key is generated for each attribute based on its current value and each node is responsible for a continuous range of values.

An important issue in large-scale resource discovery systems is load balancing. Some of the DHT-based approaches described above provide mechanisms to address this issue. For example, Bharambe et al. [2004] introduce an additional protocol to reallocate responsibilities, while Albrecht et al. [2008] propose to split the attribute space unevenly in order to uniformize the load. Although these approaches achieve the desired results, they come at the cost of increasing the complexity and overhead of the protocol. In contrast, RSS provides load balancing by design, without the need for any additional mechanisms. Most of the protocols described above have been designed to operate in environments where the number of resources largely exceeds the number of nodes. In our scenario, the published resources are the nodes themselves. Instead of having each node delegating the registration

of its attribute values to another node, each node represents itself directly in the overlay. With this approach the load is spread across nodes without any additional overhead. Our evaluation results from Section 3.3.4 show a more even distribution than using a simple DHT-based solution.

High churn rates also negatively impact DHTs [Rhea et al. 2004] and inconsistencies can arise between a resource and its representation in the DHT. In case of resource failure, a failure detector must explicitly update the DHT. Conversely, resources might become unreachable because their representative in the DHT has disconnected. In our approach, each resource represents itself in the overlay, removing the inconsistency problem: when a node's properties change, or if the node fails, no registry node must be updated. The overlay merely reconfigures to repair the broken links.

Beaumont et al. [2007] exploit Voronoi diagrams to equally partition the d-dimensional space among nodes. This approach is elegant and scales well with the number of nodes. However, as the authors note, the complexity of Voronoi diagrams is exponential with the number of dimensions, i.e., number of different attributes. This practically limits this system to supporting only two different attributes. Conversely, as we show in Section 3.3.3, our protocol scales well with the numbers of dimensions.

A requirement in many distributed applications is to have computing resources that are close to each other in terms of network latency. Albrecht et al. [2008] address this requirement by supporting inter-node parameters such as network latency and bandwidth. Although this approach offers great flexibility, it also adds a significant overhead to the search operations. Zorilla uses a flooding mechanism to identify available resources, and implements locality awareness by searching for computing nodes in the proximity of the first node that was selected for a given application [Drost et al. 2006]. While RSS does not support inter-node attributes, it can perform locality-aware searching by using network parameters that are associated with a single node (such as Vivaldi coordinates). Thus, RSS can be used to select nodes that are close in terms of network latency to a given location.

The two systems closest to our approach are [Jelasity and Kermarrec 2006] and Astrolabe [van Renesse et al. 2003]. Similar to us, both systems rely on gossip-based protocols to keep track of resources in an overlay.

Jelasity and Kermarrec [2006] propose to use gossiping to dynamically order the nodes of an overlay according to any metric such as available disk space and memory. Ordered slicing differs from our approach in two ways. First, ordered slicing is directed towards finding a *fraction* of *best* nodes in a collection. In contrast, we aim for finding any *fixed number* of *suitable* nodes. This is a different problem that cannot be easily solved through ordered slicing. Second, ordered slicing requires *all* nodes of the overlay to collaborate in answering any query. In our system, a single gossip-based overlay operates continuously in the background to maintain neighborship links between nodes. Query routing based on these links is very efficient, akin to routing in a structured peer-to-peer overlay. In other words, we separate overlay maintenance from the problem of resource selection. These two are intertwined in [Jelasity and Kermarrec 2006], so that each new query causes a rerun of the whole protocol.

In Astrolabe, nodes are organized along a tree structure. Each node gossips only with other nodes at the same level. Information about available resources is incrementally summarized as it is reported from the tree leaves toward the root. The main purpose of Astrolabe is to provide aggregated information on the status of (a part of) the system. However, reporting aggregate information is not sufficient for resource selection. As the authors acknowledge [van Renesse et al. 2003], Astrolabe can easily provide (approximate) information on how many nodes fit an application's requirements, but cannot efficiently produce the list of nodes themselves.

7.2. Self-adaptation

As we saw in Section 6, the optimal performance of the RSS requires to continuously self-adapt a number of internal system parameters. However, for query routing correctness it is essential to coordinate adaptations across the entire system such that (almost) all nodes use the same value of such parameters at any given time. To our surprise, previous work in self-adaptation in large-scale decentralized systems focuses entirely on tuning parameters at the local level rather than on "global" coordinated parameters. For example, tuning local node parameters can be used for decentralized load balancing [Steele et al. 2008], load-balancing storage and replication in DHTs [Aberer et al. 2005], adapting the critical exponent of power-law networks [Scholtes et al. 2008], and maintaining the optimal ratio of super-peers to subgroups [Sanchez-Artigas et al. 2008; Snyder et al. 2009]. Each of these is a difficult problem in itself, and one of the paramount difficulties of tuning local node parameters is to ensure that the system converges without coordination to an optimum global state. Our work instead focuses on directly adapting global parameters requiring coordination while still maintaining a completely decentralized system. To the best of our knowledge, this is the first work specifically addressing the self-adaptation of global parameters in a decentralized setting.

Our work uses a decentralized monitor for P2P systems. Some large-scale monitoring systems use hierarchical aggregation by building a tree-like topology to collect data (for example, [van Renesse et al. 2003; Yalagandula and Dahlin 2004]). However, the hierarchical topology is difficult to construct and maintain in the presence of churn, which is always present in P2P systems. A different direction builds on fully decentralized gossip-based aggregation to obtain compact statistical results like averages or total counts. This method is simple to implement, robust to churn, and provides any node in the system with the computed statistics. The compact values obtained are however not always sufficient for optimization tasks. The decentralized monitor we use in this paper provides the statistical distribution of the values of a parameter at low cost, while retaining all the advantages of other aggregation methods [Sacha et al. 2010].

8. CONCLUSIONS

Future utility computing platforms will be too large to support (semi-)centralized resource discovery. We have presented a fully decentralized protocol to select nodes according to their properties. Each node represents itself in an overlay where resource discovery queries can be routed. Self-adaptation of the routing overlay ensures efficient lookup even as the queries and composition of the system change over time.

We have shown through simulations and actual deployments that this protocol scales well with the number of nodes and dimensions. The overlay adopts a gossip-based infrastructure that continuously maintains its routing tables, making our system extremely resilient to churn. Also, no intricate measures are necessary to ensure load balancing, to recover from link or node failures, or to adapt to changes in a node's attributes. By keeping management localized and by following an autonomous approach of "continuous maintenance," our system achieves a high degree of simplicity from which the properties discussed in this paper emerge naturally.

Finally, we note that resource selection is just the first step towards a complete decentralized job execution system and other issues (e.g., scheduling [Fiscato et al. 2008], trust, and incentives schemes) deserve further investigation and are part of our future research agenda.

REFERENCES

Aberer, K., Datta, A., and Hauswirth, M. 2005. Multifaceted simultaneous load balancing in DHT-based P2P systems: A new game with old balls and bins. In *Self-star Properties in Complex Information Systems, LNCS 3460*.

Albrecht, J., Oppenheimer, D., Vahdat, A., and Patterson, D. A. 2008. Design and implementation trade-offs for wide-area resource discovery. *ACM Transactions on Internet Technology 8,* 4.

Anderson, D. P. and Reed, K. 2009. Celebrating Diversity in Volunteer Computing. In *Proceedings of the Hawaii International Conference on System Sciences*.

Anderson, T. and Roscoe, T. 2006. Learning from PlanetLab. In *Proceedings of the Workshop on Real, Large Distributed Systems*.

Beaumont, O., Kermarrec, A.-M., Marchal, L., and Rivière, E. 2007. VoroNet: A scalable object network based on Voronoi tessellations. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*.

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

Cai, M., Frank, M., Chen, J., and Szekely, P. 2003. MAAN: A multi-attribute addressable network for grid information services. In *Proceedings of the International Workshop on Grid Computing*.

Coppola, M., Jégou, Y., Matthews, B., Morin, C., Prieto, L. P., Sánchez, O. D., Yang, E., and Yu, H. 2008. Virtual organization support within a grid-wide operating system. *IEEE Internet Computing 12,* 2.

Costa, P., Napper, J., Pierre, G., and van Steen, M. 2009. Autonomous resource selection for decentralized utility computing. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*.

DAS-3. `http://www.cs.vu.nl/das3/`.

Drost, N., van Nieuwpoort, R. V., and Bal, H. 2006. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *Proceedings of the Workshop on Global and Peer-2-Peer Computing*.

Fiscato, M., Costa, P., and Pierre, G. 2008. On the Feasibility of Decentralized Grid Scheduling. In *Proceedings of the International Workshop on Decentralized Self Management For Grids, P2P, and User Communities*.

Foster, I. and Iamnitchi, A. 2003. On death, taxes, and the convergence of peer-to-peer and grid computing. In *Proceedings of the International Workshop on Peer-to-Peer Systems*.

Ganesan, P., Yang, B., and Garcia-Molina, H. 2004. One torus to rule them all: multi-dimensional queries in P2P systems. In *Proceedings of the International Workshop on the Web and Databases*.

Gupta, A., Sahin, O. D., Agrawal, D., and Abbadi, A. E. 2004. Meghdoot: Content-based publish/subscribe over P2P networks. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*.

Iosup, A., Jan, M., Sonmez, O., and Epema, D. 2007. On the dynamic resource availability in grids. In *Proceedings of the IEEE/ACM International Conference on Grid Computing*.

Iosup, A., Li, H., Jan, M., Anoep, S., Dumitrescu, C., Wolters, L., and Epema, D. H. J. 2008. The grid workloads archive. *Future Generation Computer Systems 24,* 7.

Jelasity, M. and Kermarrec, A.-M. 2006. Ordered slicing of very large-scale overlay networks. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*.

Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., and van Steen, M. 2007. Gossip-based peer sampling. *ACM Transactions on Computer Systems 25,* 3.

Montresor, A. and Jelasity, M. 2009. PeerSim: A scalable P2P simulator. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*.

Ranjan, R., Harwood, A., and Buyya, R. 2008. Peer-to-peer based resource discovery in global grids: A tutorial. *IEEE Communications Surveys and Tutorials 10,* 2.

Ratnasamy, S. et al. 2001. A scalable content addressable network. In *Proceedings of the ACM International SIGCOMM Conference*.

Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. 2004. Handling churn in a DHT. In *Proceedings of the Annual USENIX Technical Conference*.

Sacha, J., Napper, J., Stratan, C., and Pierre, G. 2010. Adam2: Reliable distribution estimation in decentralised environments. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*.

Sanchez-Artigas, M., Garcia-Lopez, P., and Skarmeta, A. F. G. 2008. On the feasibility of dynamic superpeer ratio maintenance. *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*.

Saroiu, S., Gummadi, K. P., and Gribble, S. D. 2003. Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Systems 9,* 2.

Schmidt, C. and Parashar, M. 2003. Flexible information discovery in decentralized distributed systems. In *Proceedings of the IEEE International Symposium on High-Performance Distributed Computing*.

Scholtes, I., Botev, J., Hohfeld, A., Schloss, H., and Esch, M. 2008. Awareness-driven phase transitions in very large scale distributed systems. *Proceedings of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems*.

Schütt, T. et al. 2007. A structured overlay for multi-dimensional range queries. In *Proceedings of the International Euro-Par Conference*.

Sharma, B., Chudnovsky, V., Hellerstein, J. L., Rifaat, R., and Das, C. R. 2011. Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters. In *Proceedings of the ACM Symposium on Cloud Computing*.

Snyder, P. L., Greenstadt, R., and Valetto, G. 2009. Myconet: A fungi-inspired model for superpeer-based peer-to-peer overlay topologies. *Proceedings of the IEEE International Conference on Self-Adaptive and Self-Organizing Systems*.

Spence, D. and Harris, T. 2003. Distributed resource discovery in the XenoServer open platform. In *Proceedings of the IEEE International Symposium on High-Performance Distributed Computing*.

Steele, T., Vishnumurthy, V., and Francis, P. 2008. A parameter-free load balancing mechanism for p2p networks. In *Proceedings of the International Workshop on Peer-to-Peer Systems*.

Tanin, E., Harwood, A., and Samet, H. 2007. Using a distributed quadtree in peer-to-peer networks. *The VLDB Journal 16,* 2.

van Renesse, R., Birman, K. P., and Vogels, W. 2003. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems 21,* 2.

van Steen, M. and Ballintijn, G. 2002. Achieving scalability in hierarchical location services. In *Proceedings of the International Computer Software and Applications Conference*.

Voulgaris, S. and van Steen, M. 2005. Epidemic-style management of semantic overlays for content-based searching. In *Proceedings of the International Euro-Par Conference*.

XtremLab Project. `http://xw01.lri.fr:4320/`.

Yalagandula, P. and Dahlin, M. 2004. A scalable distributed information management system. In *Proceedings of the ACM International SIGCOMM Conference*. 379–390.

Zanikolas, S. and Sakellariou, R. 2005. A taxonomy of grid monitoring systems. *Future Generation Computer Systems 21,* 1.