Autonomous Resource Selection for Decentralized Utility Computing

Paolo Costa*† Jeff Napper* Guillaume Pierre* Maarten van Steen*

*Vrije Universiteit Amsterdam
The Netherlands

†Microsoft Research Cambridge
United Kingdom

E-mail: costa@microsoft.com,{jnapper,gpierre,steen}@cs.vu.nl

Abstract

*Many large-scale utility computing infrastructures comprise heterogeneous hardware and software resources. This raises the need for scalable resource selection services, which identify resources that match application requirements, and can potentially be assigned to these applications. We present a fully decentralized resource selection 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. We show that our solution scales in the number of resources as well as in the number of attributes, while being relatively insensitive to churn and other membership changes such as node failures.*

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. We observe a steady growth of forms of utility computing where 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.

There are currently two approaches to build large-scale utility computing infrastructures. The data center approach concentrates computing power in large and powerful data centers, allowing tight control from the operators over resources. However, for very large data centers we observe *diseconomies of scale* due to increasing costs of powering and cooling the data center and acquiring the land [1].

The computing grid approach instead federalizes resources from a large number of locations [2]. Projects such as Nano Data Centers [3] and BOINC [4] reduce costs by reusing existing infrastructure (including power facilities) and may also reduce average latencies due to the geographical distribution of the resources. Recent studies show that it is possible to run long-lasting services in such environments even though resources are far less stable compared to a typical data center [5].

In a large-scale collaborative system, resource capabilities are often very diverse [6]. An essential primitive therefore consists of identifying suitable resources for each application. A 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.

A number of solutions have been proposed to tackle this problem, ranging from centralized and hierarchical node directories to DHT-based solutions [7]. All 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 three 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.

Another important aspect of resource selection services is their ability to scale, both in terms of the number of nodes they can support and in the supported number of node attributes. Indeed, utility computing platforms may 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. Centralized and hierarchical registries can in principle handle any number of attributes but have limited scalability in number of nodes, particularly in a dynamic environment where managing a robust node hierarchy is far from trivial [8]. On the other hand, DHT-based solutions usually scale very well with large number of nodes, but they do not efficiently support multidimensional-range searches.

This paper introduces a decentralized resource selection service where each compute node is solely responsible for its own attributes. Eliminating delegation represents a simple solution to both implement efficient lookups and support large dimensionality data. 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.

In our solution each node is 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 so that queries can be forwarded to a node that either lies in the associated subregion or to a neighboring node closest to that region. Once a query is being processed within the associated subregion, it needs merely be forwarded to enough nodes within that subregion to satisfy the query. Epidemic protocols [9] manage the overlay network, enabling high resilience to faults with negligible overhead.

We evaluate our system using both simulations (with infrastructures consisting of up to 100,000 nodes), emulations (of up to 1,000 nodes running on a cluster) and actual deployment on PlanetLab (up to 300 nodes). All experiments show that the system can support high churn and scales extremely well in both the number of nodes and the number of dimensions. Notably this last property is rarely satisfied by other decentralized solutions. Our solution also provides very good balancing of the query load, even in the presence of skewed query patterns.

2. Related Work

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 [10]. While these solutions are well-suited for clusters or small collections of PCs, they exhibit scalability issues when the system size increases [11]. Hence, in recent years, researchers put forth a large effort to devise decentralized solutions addressing large-scale scenarios [12].

The vast majority of these systems exploit DHTs to map resources to nodes in order to distribute the search operations across different nodes [7]. Early approaches maintain a separate DHT per attribute: a query is executed in parallel on every overlay network and results are then intersected [13], [14]. 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 [15], [16]. More recent work, inspired by CAN [17], partitions the d-dimensional space into smaller blocks that are assigned to specific nodes, which are responsible for all resources falling in that block [18], [19], [20], [21]. Finally, SWORD [22] generates a different DHT key for each attribute based on its current value and each node is responsible for a continuous range of values.

The fundamental difference with our approach is that these protocols 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. As discussed above and evaluated extensively in Section 6.4, in contrast to DHT-based solutions, our system addresses load balancing by design, since each node manages exactly one resource (i.e., itself). Some DHT-based approaches (e.g., [18]) periodically run an additional protocol to reallocate responsibilities. This, however, comes at the cost of increasing the complexity and overhead of the protocol without providing any strong load-balancing guarantees.

High churn rates also negatively impact DHTs [23] 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. exploit Voronoi diagrams to equally partition the d-dimensional space among nodes [24]. 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 6.3, our protocol scales well with the numbers of dimensions.

Zorilla is a resource discovery system based on an unstructured overlay, resembling the Gnutella network [25]. This approach relies on message flooding to identify available resources, thus hampering its scalability. Also, this system does not support attribute-based resource selection as in our approach.

The two systems closest to our approach are [26] and Astrolabe [27]. Similar to us, both systems rely on gossip-based protocols to keep track of resources in an overlay.

Jelasity and Kermarrec propose to use gossiping to dynamically order the nodes of an overlay according to any metric such as available disk space and memory [26]. 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 neighboring 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 [26], 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 [27], Astrolabe can easily provide (approximate) information on how many nodes fit an application's requirements, but cannot efficiently produce the list of nodes themselves.

3. System Model

In our model, each node is characterized by a set of (*attribute*,*value*) pairs such as memory, bandwidth and CPU power. For sake of simplicity, we assume that the number of attributes is fixed and known *a priori*. We also assume that attribute values can be uniquely mapped to natural numbers (although they need not be represented as such).

We represent the overlay as a $d-$dimensional space $\mathcal{A} \triangleq \mathcal{A}_0 \times \mathcal{A}_1 \times \cdots \times \mathcal{A}_{d-1}$, 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 $(v_0, v_1, \ldots, v_{d-1})$, and v_i being 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. 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 subspace $\mathcal{Q}(q) \triangleq \mathcal{Q}_0 \times \mathcal{Q}_1 \times \cdots \times \mathcal{Q}_{d-1}$, 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. Ignoring strict notational issues, an example query could then be formulated as:

CPU	$=$	IA32
MEM	$\in$	$[4GB, \infty)$
BANDWIDTH	$\in$	$[512Kb/s, \infty)$
DISK	$\in$	$[128GB, \infty)$
OS	$\in$	{Linux 2.6.19-1.2895, …, Linux 2.6.20-1.2944}

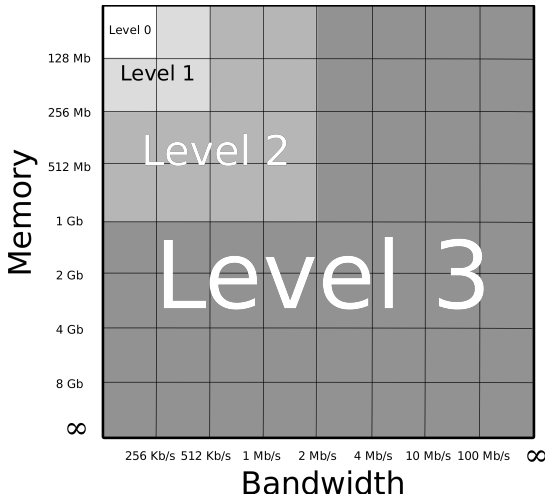
A query can be issued at any node; there is no designated node where queries should initially be sent to. Finally we assume that the network is fully connected: each node can reach any other node.

4. Protocol Description

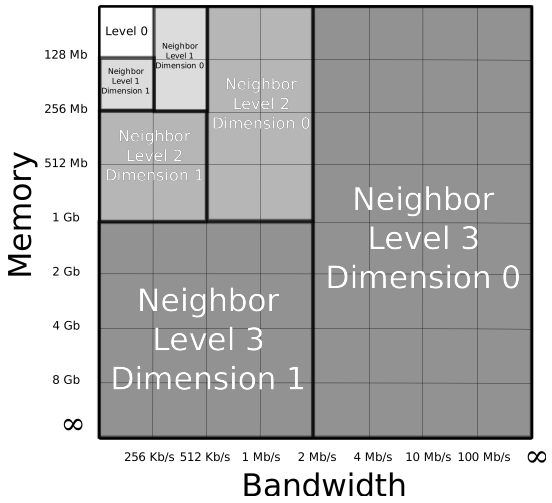
This section illustrates the resource discovery protocol. We first describe the properties of the overlay, then detail query routing. The next section discusses how the overlay is built and maintained over time.

4.1. Overlay Network Topology

The model described in Section 3 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



(a) Nested Cells.



(b) Neighboring Cells

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

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

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.

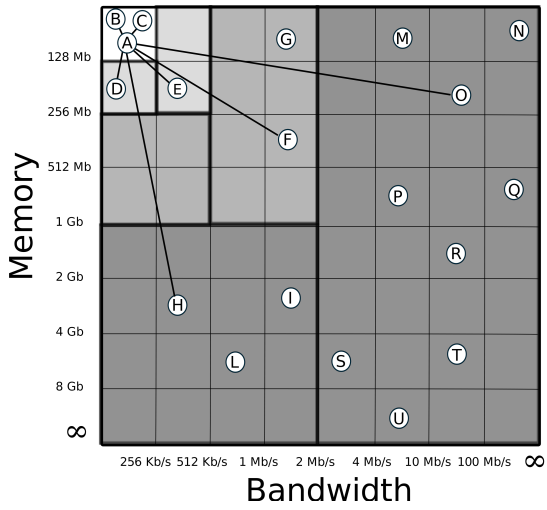
To distinguish among cells, we introduce the notion of level l. The smallest cells are at level zero. These are denoted as $\mathcal{C}_0$. $\mathcal{C}_1$ cells are obtained by grouping four $\mathcal{C}_0$ cells. Similarly, four $\mathcal{C}_1$ cells create a single $\mathcal{C}_2$ cell and so on. 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 $\mathcal{C}_l(X)$.

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 $\mathcal{C}_l(X) \setminus \mathcal{C}_{l-1}(X)$. To this end, 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.

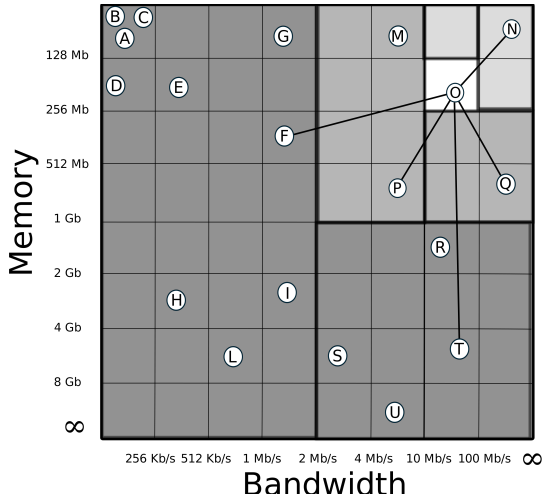
We require that a node knows one other *neighbor* node falling in one of these subcells for each level $l > 0$. If no node is present in a given subcell, then no link must be maintained. The nodes in $\mathcal{C}_0(X)$ are arranged in such a way that X can efficiently broadcast a message to each of them, for example through an epidemic protocol [28]. 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 $n_{(l,k)}(X)$. Interestingly, 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 sake of clarity, we omit the connections among the other nodes). First, A is connected with all the 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 that no link is created. The same procedure is repeated for $l = 3$ (nodes O and H are selected). Similarly, Figure 2(b) reports the links of node O. Links need not be symmetric. For instance, here O is a neighbor of A but not vice versa.

Note that even if the nodes are not uniformly distributed throughout the space, the performance of our protocol is not affected. In fact, our protocol can even benefit to a certain extent 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



(a) Node A's neighbors



(b) Node O's neighbors

Figure 2. Neighbor links for nodes A and O.

cells will be empty (e.g., A has no neighbor in $\mathcal{N}_{(2,1)}(A)$). In addition, highly populated cells avoid bottlenecks by offering more nodes from which to select a neighbor.

4.2. Query Routing: An Example

We illustrate query routing by an example depicted in Figure 3. 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 3, 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

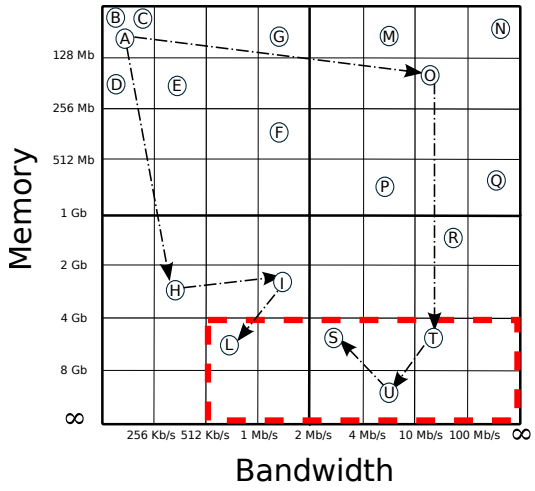


Figure 3. Example of query routing.

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[1].

As shown in Figure 3, 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. 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.

4.3. Query Routing: The Protocol

When a user needs a collection of nodes to perform her tasks, she contacts any node in the overlay network and passes the query to it. A query message contains the address

1. 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.

QUERY:
- *id*: the query identifier (must be unique)
- *address*: the address of the last forwarder of the query
- *ranges*: the vector of the desired ranges per attribute
- σ: the number of nodes to be found (optional)
- *level*: the cell level to explore. Default value is $\max(l)$.
- *dimensions*: the set of dimensions to explore. Default value is $\{0, 1, \ldots, d-1\}$.

REPLY:
- *id*: the id of the corresponding query
- *matching*: the set of nodes (address, values) matching the query
- *sender*: the address of reply's sender

(a) Message Formats.

Each node X hosts a set $neighbors$, containing one neighbor $n_{(l,d)}(X)$ per neighboring cell at level l and dimension d. Neighbors in $\mathcal{C}_0(X)$ are kept in a separate set $neighborsZero$.
For each neighbor the following information is stored:
- *n.address*: n's IP address and TCP port
- *n.level*: the level of this neighbor
- *n.dimension*: the dimension of this neighbor

A node also maintains the following vectors (maps) indexed by the query id:
- *pending*: the queries that have not been answered yet
- *matching*: the addresses of the nodes matching the query collected so far
- *waiting*: the addresses of the nodes to which the query has been forwarded but from which no reply has been received yet

Each node stores information about its own state in a record *self* with fields:
- *address*: own IP address and TCP port
- *values*: a vector of all its attribute values

We also rely on the following Boolean functions:
- *overlaps(q, l, d, X)*: returns TRUE if $\mathcal{Q}(q)$ overlaps $\mathcal{N}_{(l,d)}(X)$
- *matches(n,q)*: returns TRUE if node n's attributes match the query q

(b) Data structures.

Figure 4. Messages and data structures.

of the querying node and the desired range of values for each attribute representing a lower and upper bound. The job may specify both of them, only one, or even none (if it does not care for any particular value for that attribute).

A job can also impose an upper bound σ on the number of nodes it is interested in. This information will be used by our protocol to halt the propagation of the query once the threshold is met. Finally, each query contains two further fields, *level* and *dimensions*, which will be exploited to forward the query. Default values are reported in Figure 4(a).

Each node stores three tables containing an entry for each query it receives (Figure 4(b)). The first one, *pending*, is used to keep track of on-going queries. Each entry is associated with a time out $T(q)$: when it expires, the neighbor is considered to have failed and the query is forwarded again. The second table, named *matching*, includes the list of candidates that the node has retrieved so far for each query in *pending*. Finally, *waiting* stores the neighbors that have received a query but have not replied yet.

As shown in the second procedure in Figure 5, when a node X receives a query, it first adds a new entry in the three aforementioned tables (lines 1–3). It then checks whether its own attributes satisfy the request, in which case it adds itself to the *matching* list (lines 4–5). It then invokes the `forward` procedure to route the query to its neighbors (lines 6–7), unless σ nodes have already been found.

In the `forward` procedure, all neighboring cells are

Invoked by a user to query for a collection of k nodes.
`create QUERY` q
1: create a QUERY message q
2: $q.address \leftarrow self.address$
3: **for all** $\mathcal{A}_i \in \mathcal{A}$ **do**
4: $q.ranges[i] \leftarrow (min_i, max_i)$
5: $q.\sigma \leftarrow k$
6: $q.level \leftarrow max(l)$
7: $q.dimensions \leftarrow \{0, 1, \ldots, d-1\}$
8: receive_query(q)

Invoked by a node receiving a QUERY message.
`receive_query QUERY` q
1: $pending[q.id] \leftarrow q$
2: *matching*$[q.id] \leftarrow \varnothing$
3: *waiting*$[q.id] \leftarrow \varnothing$
4: **if** *matches(self, q)* **then**
5: *matching*$[q.id] \leftarrow \{self\}$
6: **if** $|$*matching*$[q.id]| < q.\sigma$ **then**
7: forward(q)
8: **else**
9: create REPLY r
10: $r.sender \leftarrow self.address$
11: $r.id \leftarrow q.id$
12: r.*matching* $\leftarrow$ *matching*$[q.id]$
13: **send** $r \rightarrow sender[q.id]$

Invoked by a node to forward a QUERY message.
`forward QUERY` q
1: **while** $q.level > 0$ **do**
2: **for all** $d' \in q.dimensions$ **do**
3: **if** *overlaps(q, q.level, d', self)* **then**
4: $q.dimensions \leftarrow q.dimensions \setminus \{d'\}$
5: **send** $q \rightarrow neighbors[q.level, d']$
6: *waiting*$[q.id] \leftarrow$ *waiting*$[q.id] \cup \{neighbors[q.level, d']\}$
7: **return**
8: $q.level \leftarrow q.level - 1$
9: $q.dimensions \leftarrow \{0, 1, \ldots, d-1\}$
10: **if** $q.level = 0$ **then**
11: **for all** $n \in neighborsZero$ **do**
12: **if** *matches(n, q)* $\wedge\ n \notin$ *matching*$[q.id]$ **then**
13: $q' \leftarrow q$
14: $q'.level \leftarrow -1$
15: **send** $q' \rightarrow n$
16: *waiting*$[q.id] \leftarrow$ *waiting*$[q.id] \cup \{n\}$
17: **return**
18: **if** $|$*waiting*$[q.id]| = 0$ **then**
19: create REPLY r
20: $r.sender \leftarrow self.address$
21: $r.id \leftarrow q.id$
22: r.*matching* $\leftarrow$ *matching*$[q.id]$
23: **send** $r \rightarrow sender[q.id]$

Invoked by a node receiving a REPLY message.
`receive_reply REPLY` r
1: $q \leftarrow pending[r.id]$
2: *matching*$[r.id] \leftarrow$ *matching*$[q.id] \cup r$.*matching*
3: *waiting*$[q.id] \leftarrow$ *waiting*$[q.id] \setminus \{r.address\}$
4: **if** $|$*waiting*$[q.id]| \neq 0$ **then**
5: **return**
6: **else if** $|$*matching*$[r.id]| < q.\sigma \wedge q.level \geq 0$ **then**
7: forward(q)
8: **else**
9: create REPLY r'
10: $r'.sender \leftarrow self.address$
11: $r'.id \leftarrow r.id$
12: r'.*matching* $\leftarrow$ *matching*$[r.id]$
13: **send** $r' \rightarrow q.address$

Figure 5. Query routing protocol.

scanned sequentially starting from the highest level until a cell overlapping $\mathcal{Q}$ is found In this case, the query message is forwarded to the neighbor responsible for that cell and, after storing its address in *waiting* list, the procedure terminates (lines 1–7). To prevent this neighbor from sending the query back, the corresponding dimension of the neighboring cell is removed from the query (line 4). This way, when the neighbor wants to forward the query, it will be prevented from sending the message along the same dimension. However, when passing to a lower level, dimensions are reset to the default value (lines 8–9).

If no suitable neighbor is found, nodes from $\mathcal{C}_0(X)$ (i.e., the $neighborsZero$ set) are checked to verify whether matches exist (lines 10–17). Finally, if the query could not be forwarded, X replies to the node from which it received the query by sending the set of matching nodes. This set can either be empty or contain X itself (lines 18–23).

Upon the receipt of a REPLY message, a node retrieves the corresponding query from the *pending* list, adds the addresses of the nodes included in the reply to its own *matching* list, and deletes the reply's sender from the *waiting* list (lines 1–3). Then, if there is still some neighbor that has not replied yet ($|$*waiting*$[q.id]| \neq 0$), the procedure terminates (lines 4–5). Otherwise, if the number of candidates found so far is still lower than what was initially required and there are still some neighboring cells to explore (i.e., $q.level \geq 0$), the forwarding procedure is invoked again (lines 6–7). If enough candidates have been found, a new reply message is created and filled with the addresses of the discovered candidates and sent back to the node from which the query message had been received (lines 8–13).

5. Overlay Maintenance

An important issue in the above protocol is to efficiently maintain the overlay in the presence of frequent node joins and leaves, for example caused by failures. Similarly, node attributes might change during a system's lifetime, introducing another source of dynamicity.

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 two-layered gossip-based protocol [9]. The bottom layer executes the CYCLON protocol [28], [29] 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 second gossip-based layer executes a protocol very similar to the first one in that each node keeps another 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, in the second layer links are associated with the attribute values of the node they represent. Nodes do not randomly select links to keep in their list, but according

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)	5
Nesting depth ($max(l)$)	3
Gossip period	10 seconds
Gossip cache size	20

Table 1. Default simulation parameters.

to their attributes. Specifically, each node X selects only links to nodes located in its neighboring cells $\mathcal{N}_{(l,k)}(X)$.

As discussed in [9] and Section 6.6, this approach for self-organization converges extremely 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. At the same time, the underlying CYCLON layer continuously feeds the top layer with random nodes to make sure the system remains connected.

6. Evaluation

To assess the performance of our protocol, we built two implementations. We deployed the first implementation on the DAS-3 cluster at VU University Amsterdam [30]. 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 [31]. This allows us to explore setups with up to 100,000 nodes. Finally, we run our system on 302 nodes in PlanetLab [32].

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 traveled 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. We discuss the effects of churn on delivery in Section 6.6. In addition, in all runs (including the ones with churn), a message has never been received twice by the same node.

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.

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 $f \times N$ nodes. Different queries refer to different subspaces. Each query is then issued repeatedly from every node in the system. Unless specified otherwise, simulations are based on the default parameters depicted in Table 1.

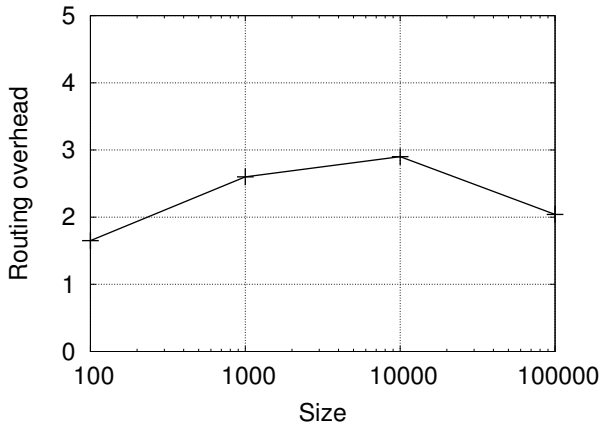


Figure 6. Routing overhead vs. network size (PeerSim).

Hereafter we focus on the performance of the protocol and ignore costs due to the maintenance of the overlay. These costs depend on the gossip frequency: for each gossip cycle, each node initiates exactly two gossips (one per gossip layer), and receives on average two other gossips. With message sizes of 320 bytes, this yields a traffic of 2,560 bytes per gossip cycle at each node. Given a gossip periodicity of 10 seconds, we consider these costs as negligible.

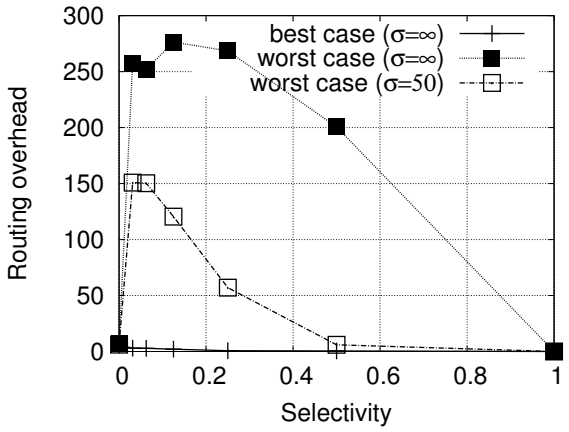
6.1. Effect of Network Size

Figure 6 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. Interestingly, 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.

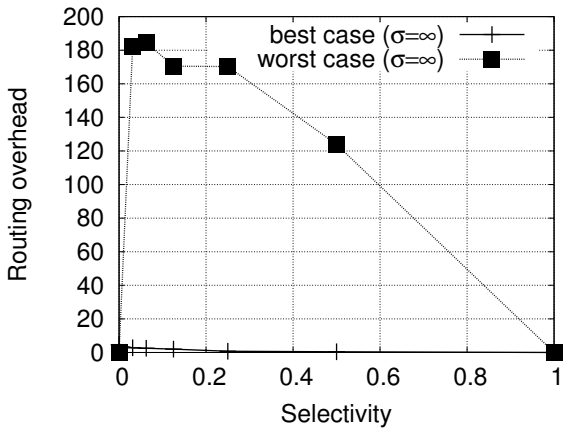
6.2. Effect of Query Selectivity

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 requires to route the query on every dimension and level, thus increasing the path to reach all matching nodes[2].

2. In practice, cell boundaries can be set to specific values. We can also force queries to respect boundaries in order to reduce the likelihood that a query spans multiple subcells. For example, an application in need of 1.2–2.9 GB of memory may be forced to request 1–3 GB.



(a) PeerSim



(b) DAS

Figure 7. Routing overhead vs. selectivity.

Figure 7 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 7(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.

Interestingly, the overhead in the worst case does not change significantly between 100,000 (Figure 7(a)) and 1,000 nodes (Figure 7(b)). Indeed, 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.

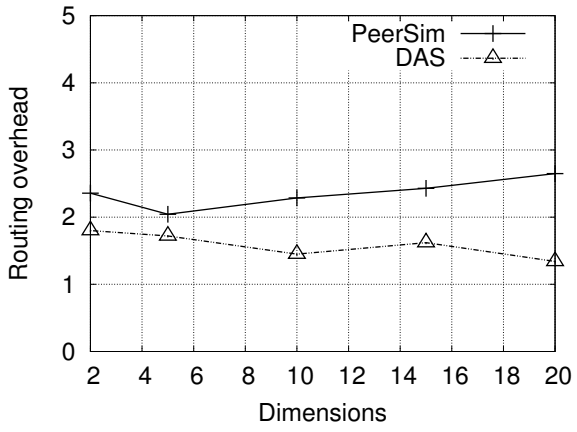


Figure 8. Routing overhead vs. dimensions.

6.3. 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 8 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.

6.4. Load Distribution

In a large-scale system, it is important that the load imposed by the protocol is evenly distributed among nodes. Figure 9(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 gossip-based construction of the neighbor lists. 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 9(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 are taken from the XtremLab BOINC project traces [33] that record node properties seen for more than 10,000 hosts in BOINC projects and are highly skewed. We use the Bamboo DHT [23] and, as in SWORD [22], store a record of the nodes' attributes in the DHT at a key for each attribute value for each dimension. In each run we randomly generated 50 queries ($f = 0.125$) and measured the messages processed by each node. Searches are performed using a range query (implemented as an iterated search) 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[3].

6.5. Number of Neighbors per node

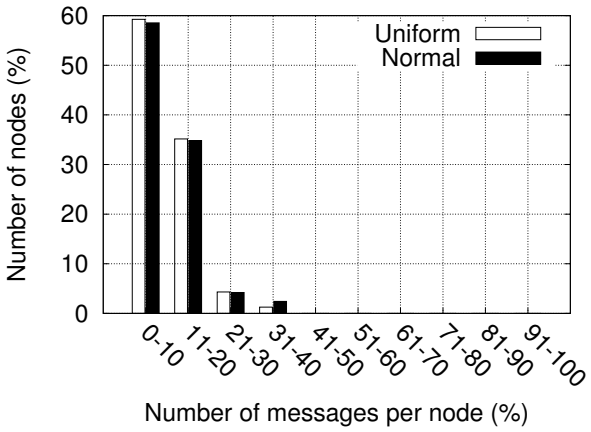
The next evaluation concerns the number of links that each node must maintain. Links belong to two categories. First, a node must maintain its *neighborZero* list which links to every other node present in the same lowest-level cell. The number of cells in the system is $\left(2^d\right)^{\max(l)}$, 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 [28].

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 10(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[4]. Similar results (omitted here for space reasons) are also obtained when varying $\max(l)$.

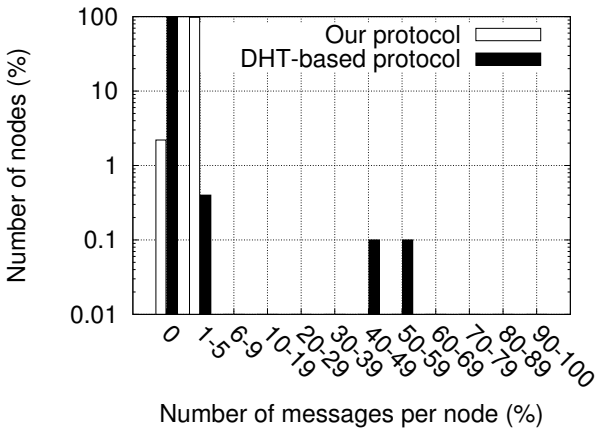
Figure 10(b) plots the distribution of the of links per node in PeerSim, under uniform and normal distribution. In both cases, this number remains under 20 links in total. We note, however, that the normal distribution case requires slightly more links per node. This is due to the fact that *neighborZero* lists will grow in the cells around the hotspot.

3. 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.

4. For $d < 5$ the number of neighbors maintained by each node is bounded by the gossip cache (equal to 20 in our configuration).



(a) Uniform vs. normal distribution



(b) Ours vs. DHT-based approach

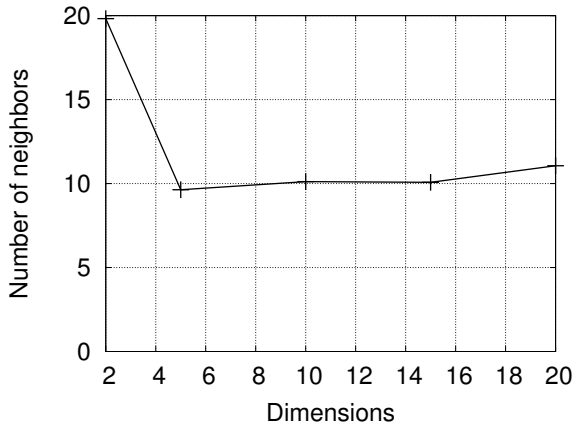
Figure 9. Node load distribution

6.6. Delivery under Churn

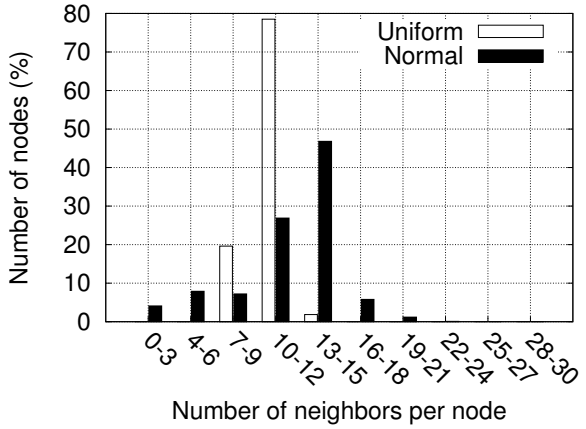
Experiments presented so far assume that the list of nodes remains stable. This is of course unrealistic: any large network will exhibit a degree of dynamicity due to node joins and leaves. In particular, ungraceful node departures may represent an issue, since the routing tables of other nodes need 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 evaluated the *delivery* (i.e., the fraction of matching nodes that actually receive the query) in PeerSim when 0.1% (resp. 0.2%) of the nodes leave the system and re-enter it under a different identity every 10 seconds. The 0.2% value corresponds to churn rates observed in Gnutella [34]. However, many real-world systems are considerably more stable [35].

We measure the delivery over time by issuing one query every 30 seconds. Queries do not use any threshold value, so a delivery of 1 means that we reached all $f \times N = 12,500$

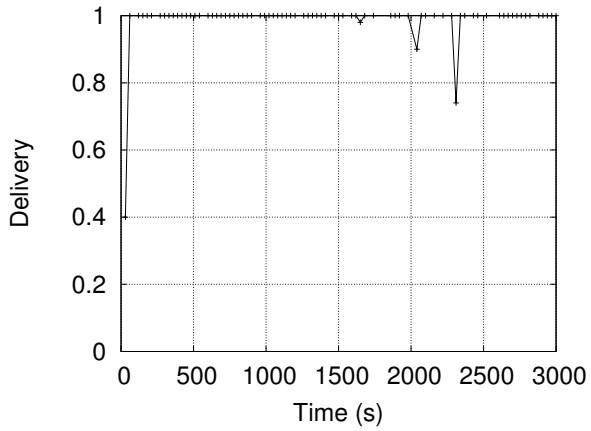


(a) Neighbors vs. dimensions

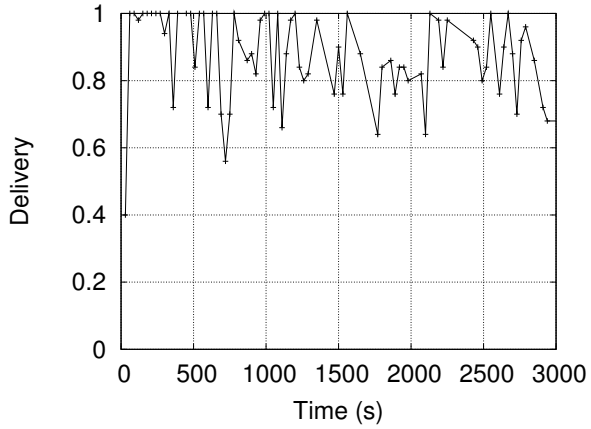


(b) Neighbors distribution

Figure 10. Number of neighbors (PeerSim)



(a) Churn = 0.1% per $10s$.



(b) Churn = 0.2% per $10s$.

Figure 11. Delivery vs. churn (PeerSim).

matching nodes. As shown in Figure 11, a churn of 0.1% barely disrupts the delivery. With 0.2% churn the delivery decreases, but remains still high. Note that a delivery of 0.8 means that we retrieved 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 $N = 100,000$ and a selectivity $f = 0.125$, a delivery of 0.8 yields around 10,000 nodes (i.e., $N \cdot f \cdot 0.8$), which is well-above the expected number of nodes needed for a job.

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. While we did not adopt this approach to avoid any bias, this would have allowed delivery close to 1. Latency would have increased though, because nodes, upon the detection of a failed neighbor, would wait for the overlay to be repaired before forwarding the query.

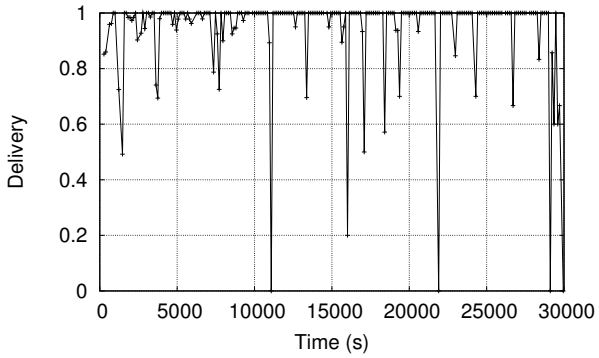
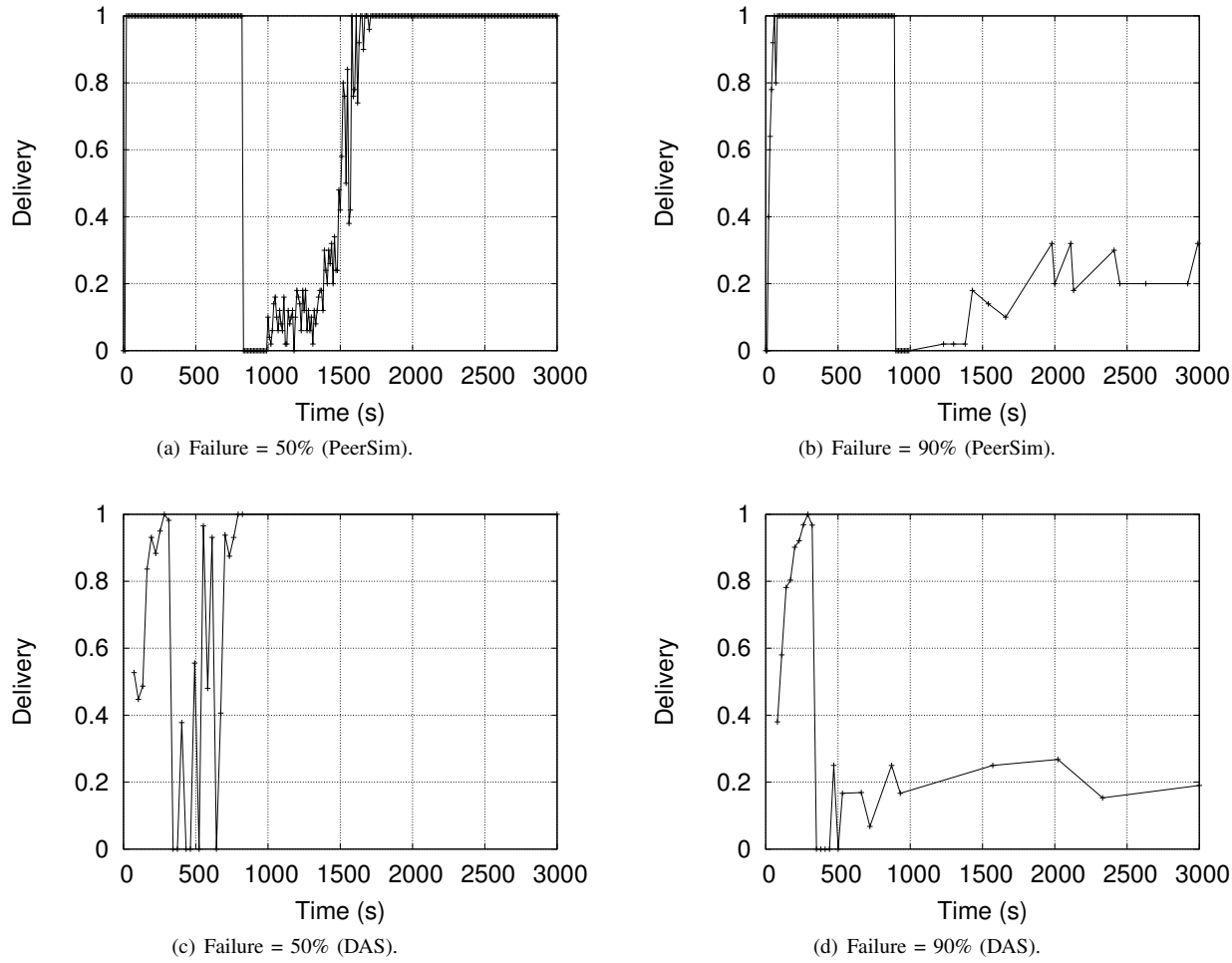


Figure 13. Delivery vs. repeated massive failures (PlanetLab).

6.7. Delivery under Massive Failure

The last 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



(a) Failure = 50% (PeerSim).

(b) Failure = 90% (PeerSim).

(c) Failure = 50% (DAS).

(d) Failure = 90% (DAS).

Figure 12. Delivery vs. massive failure.

failure. Again, we do not use any threshold values or the previously discussed mechanism to avoid evaluation bias.

Figure 12(a) and 12(b) shows delivery in PeerSim when we remove respectively 50% and 90% of random nodes from the network at once. We evaluate the delivery once every 10 seconds. 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 only 15 minutes to recover completely. This value may be tuned by changing the gossip period. Only in the case 90% simultaneous failures, the delivery could not be restored. The overlay was partitioned by the massive failure so full recovery was impossible. Similar behaviors are also observed in Figure 12(c) and 12(d) in the DAS setup.

To evaluate our protocol on a wide area network, we deployed it on 302 nodes in PlanetLab. We tested it under very challenging conditions, artificially increasing the natural churn of PlanetLab by killing 10% of the network every 20 minutes. These nodes were not replaced, so the system shrinks over time. Figure 13 again shows fast recovery and near-optimal delivery once the routes have been restored.

7. 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.

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 which 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 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 systems and other issues (e.g., scheduling [36], trust, and incentives schemes) deserve further investigation and are part of our future research agenda.

References

[1] K. Church, A. Greenberg, and J. Hamilton, "On Delivering Embarassingly Distributed Cloud Services," in *Proc. Workshop on Hot Topics in Networking (HotNets-VII)*, 2008.

[2] H. Stockinger, "Defining the Grid: A Snapshot on the Current View," *Journal of Supercomputing*, vol. 42, 2007.

[3] N. Laoutaris, P. Rodriguez, and L. Massoulie, "ECHOS: Edge Capacity Hosting Overlays of Nano Data Centers," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 1, 2008.

[4] D. P. Anderson, C. Christensen, and B. Allen, "Designing a runtime system for volunteer computing," in *Proc. Supercomputing Conference (SC)*, 2006.

[5] A. Andrzejak, D. Kondo, and D. P. Anderson, "Ensuring Collective Availability in Volatile Resource Pools Via Forecasting," in *Proc. Intl. Workshop on Distributed Systems: Operations and Management (DSOM)*, 2008.

[6] D. P. Anderson and K. Reed, "Celebrating Diversity in Volunteer Computing," in *Proc. Hawaii Intl. Conference on System Sciences (HICSS)*, 2009.

[7] R. Ranjan, A. Harwood, and R. Buyya, "Peer-to-peer based resource discovery in global grids: A tutorial," *IEEE Communications Surveys and Tutorials*, vol. 10, no. 2, 2008.

[8] M. van Steen and G. Ballintijn, "Achieving scalability in hierarchical location services," in *Proc. Intl. Computer Software and Applications Conference (CompSac)*, 2002.

[9] S. Voulgaris and M. van Steen, "Epidemic-style management of semantic overlays for content-based searching." in *Proc. Euro-Par Conference*, 2005.

[10] S. Zanikolas and R. Sakellariou, "A taxonomy of grid monitoring systems," *Future Generation Computer Systems*, 2005.

[11] T. Anderson and T. Roscoe, "Learning from PlanetLab," in *Proc. Workshop on Real, Large Distributed Systems*, 2006.

[12] I. Foster and A. Iamnitchi, "On death, taxes, and the convergence of peer-to-peer and grid computing," in *Proc. Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[13] D. Spence and T. Harris, "Distributed resource discovery in the XenoServer open platform," in *Proc. HPDC*, 2003.

[14] M. Cai, M. Frank, J. Chen, and P. Szekely, "MAAN: A multi-attribute addressable network for grid information services," in *Proc. Intl. Workshop on Grid Computing*, 2003.

[15] P. Ganesan, B. Yang, and H. Garcia-Molina, "One torus to rule them all: multi-dimensional queries in P2P systems," in *Proc. Intl. Workshop on the Web and Databases*, 2004.

[16] C. Schmidt and M. Parashar, "Flexible information discovery in decentralized distributed systems," in *Proc. HPDC*, 2003.

[17] S. Ratnasamy *et al.*, "A scalable content addressable network," in *Proc. SIGCOMM*, 2001.

[18] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting scalable multi-attribute range queries," in *Proc. SIGCOMM*, 2004.

[19] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi, "Meghdoot: Content-based publish/subscribe over P2P networks," in *Proc. Intl. Middleware Conference*, 2004.

[20] T. Schütt, F. Schintke, and A. Reinefeld, "A structured overlay for multi-dimensional range queries," in *Proc. Euro-Par Conference*, 2007.

[21] E. Tanin, A. Harwood, and H. Samet, "Using a distributed quadtree in peer-to-peer networks," *VLDB Journal.*, 2007.

[22] J. Albrecht, D. Oppenheimer, A. Vahdat, and D. A. Patterson, "Design and implementation trade-offs for wide-area resource discovery," *ACM Trans. Interet Technol.*, vol. 8, no. 4, 2008.

[23] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," in *Proc. USENIX Technical Conf.*, 2004.

[24] O. Beaumont, A.-M. Kermarrec, L. Marchal, and E. Rivière, "VoroNet: A scalable object network based on Voronoi tessellations," in *Proc. IPDPS*, 2007.

[25] N. Drost, R. V. van Nieuwpoort, and H. Bal, "Simple locality-aware co-allocation in peer-to-peer supercomputing," in *Proc. Work. on Global and Peer-2-Peer Computing*, 2006.

[26] M. Jelasity and A.-M. Kermarrec, "Ordered slicing of very large-scale overlay networks," in *Proc. of P2P*, 2006.

[27] R. V. Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Transactions on Computer Systems*, vol. 21, no. 2, 2003.

[28] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems*, vol. 25, no. 3, 2007.

[29] S. Voulgaris, D. Gavidia, and M. van Steen, "CYCLON: Inexpensive membership management for unstructured P2P overlays." *Jrnl. of Network and Systems Management*, 2005.

[30] DAS-3, http://www.cs.vu.nl/das3/.

[31] PeerSim, http://peersim.sourceforge.net.

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

[33] XtremLab Project, http://xw01.lri.fr:4320/.

[34] S. Saroiu, K. P. Gummadi, and S. D. Gribble, "Measuring and analyzing the characteristics of Napster and Gnutella hosts," *Multimedia Systems*, vol. 9, no. 2, 2003.

[35] A. Iosup, M. Jan, O. Sonmez, and D. Epema, "On the dynamic resource availability in grids," in *Proc. Conf. on Grid Computing (GRID)*, 2007.

[36] M. Fiscato, P. Costa, and G. Pierre, "On the Feasibility of Decentralized Grid Scheduling," in *Proc. Intl. Workshop on Decentralized Self Management For Grids, P2P, And User Communities (SelfMan)*, 2008.