A Self-Organizing P2P System with Multi-Dimensional Structure

Raffaele Giordanelli, Carlo Mastroianni
ICAR-CNR, Rende(CS), Italy
{giordanelli,mastroianni}@icar.cnr.it

Michela Meo
Politecnico di Torino, Italy
michela.meo@polito.it

ABSTRACT

This paper presents and analyzes Self-CAN, a self-organizing P2P system that, while relying on the multi-dimensional structured organization of peers provided by CAN, exploits the operations of ant-based mobile agents to sort the resource keys and distribute them to peers. The benefits of the self-organization approach are remarkable, starting from increased flexibility and robustness, to better load balancing characteristics. Most notably, peer indexes and resource keys can be defined on different and independent spaces, which overcomes the main limitation of standard structured P2P systems, i.e., the necessity of assigning each key to a peer having a specified index. This decoupling opens the possibility of giving a semantic meaning to resource keys and enables the efficient execution of multi-dimensional range queries, which are essential in some types of distributed systems, for example in Grids.

Categories and Subject Descriptors

H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software—*Distributed systems, Performance evaluation (efficiency and effectiveness)*

General Terms

Algorithms, Design

Keywords

Bio-Inspired, Key-value Storage, Peer-to-peer, Resource Discovery, Self-organizing

1. INTRODUCTION

Peer-to-peer (P2P) techniques and algorithms are steadily emerging as efficient solutions for the management of large-scale distributed computing systems. In particular, the P2P paradigm is used to cope with the placement, advertising and discovery of resources needed by users for the execution of complex applications. P2P presents significant advantages over centralized and hierarchical solutions, which may suffer from scalability and fault tolerance problems. Besides traditional file-sharing applications, new fields of application for P2P systems are Grid and Cloud Computing.

In Grids [1], different types of computing resources need to be shared among the nodes of a community, ranging from storage and processing devices to data, programs, and software facilities. In this context, resources are not typically discovered through their names, as in file sharing systems, but through a set of characteristics, which are described by resource metadata documents or "profiles". P2P solutions can also be exploited to manage the information systems of companies, like Google, Amazon and Microsoft, which offer pay-as-you-go-services through their Cloud platforms [2]. For example, Amazon Dynamo [3], a key-value storage system adopted by several Amazon's core services, is a variant of the the Chord P2P system.

Most modern P2P architectures are "structured", which means that the resource keys, through which resource profiles can be discovered and accessed, are assigned to nodes with a predetermined strategy, and using *Distributed Hash Tables*, or DHTs. When a resource is published, its access key is computed with a hash function applied to the resource name; then, the resource profile is assigned to the node that is responsible for that key. The keys are assigned to nodes according to different types of structures, such as rings, multi-dimensional grids or trees: these structures are used, respectively, by Chord [4], CAN [5], and Pastry [6]. The main benefits of structured systems is that they can use "informed" algorithms to drive user queries towards the desired keys in a short and bounded time, and generate low network traffic. A major drawback, however, is that the semantic profile of the resource cannot be used in the discovery process. Therefore it is not possible to efficiently serve complex queries, or range queries that aim at finding resources sharing common features. Range queries are particularly important in Grid environments: a user often needs to discover a set of resources that are compatible with the application s/he wants to run, for example, a set of machines with CPU speed and RAM memory comprised in a given range. Other drawbacks concern the load balance (the nodes that are assigned the most popular keys may be highly loaded) and the dynamic behavior of the system (for example, an immediate reassignment of keys is necessary every time a node joins or leaves the system).

In recent years, there have been interesting attempts to reinforce the adaptive and fault-tolerance characteristics of

ICAC'11, June 14–18, 2011, Karlsruhe, Germany.

P2P networks by imitating the self-organizing behavior of biological systems, such as flocks of birds, insect swarms, and, above all, ant colonies [7, 8]. These algorithms exploit the properties of "swarm intelligence" systems, in which an intelligent behavior at a high level is obtained by combining simple low level operations performed by bio-inspired mobile agents [9]. Self-Chord [10] exploits the structured ring-shaped overlay offered by Chord, and sorts the keys over the ring without any predefined association between peer codes and keys. This allows resource keys to assume a semantic meaning, and facilitates the execution of range queries. These advantages, though, are limited to the cases where a single attribute can be used to characterize the resources, which is often a too strict constraint.

This paper shows that it is possible to use ant-inspired algorithms to manage multi-attribute resource profiles, and map them to a multi-dimensional structure, specifically the one offered by CAN. Accordingly, the system presented here is referred to as "Self-CAN". In the CAN base system, each node that joins the network is assigned an index, randomly computed through a hash function, which corresponds to a point in a multi-dimensional logical space. This node is then declared responsible for a zone of the space that includes its index, and manages the resource keys that belong to the same space. In Self-CAN, the spaces of resource keys and node indexes are decoupled, so that a key does not need to be assigned to a specified node. Instead, ant-inspired agents sort the multi-dimensional resource keys over the CAN structure through a self-organizing algorithm. The fundamental consequence is that in Self-CAN resource keys do not have to be calculated with a hash function, as in CAN, but can preserve the semantic meaning of resource attributes. For example, a three-dimensional key of a data mining service can express the degree of availability, the cost, and the type of algorithm executed by the service.

Search messages can be driven to the desired keys in logarithmic time, by following the gradient of the keys stored by different nodes. Moreover, thanks to the sorting of keys over the structured overlay, Self-CAN is particularly efficient in the execution of complex and multi-dimensional range queries. Notice that such queries are inefficient in CAN, since the keys of similar resources are uniformly spread by the hash function in the multi-dimensional structure.

A prototype of Self-CAN, written in Java, is available on the Web site http://self-can.icar.cnr.it. Performance evaluation was carried out either with the prototype or with an event-based simulator that emulates the behavior of the prototype, depending on the size of the evaluated network. The paper is organized as follows: Section 2 introduces the Self-CAN model and illustrates the effect of the sorting process; Section 3 describes the operations performed by the ant-inspired agents; the results of a wide set of experiments are discussed in Section 4, conveniently divided in subsections; finally, after the related work section, Section 6 concludes the paper.

2. THE SELF-CAN MODEL

Self-CAN uses the same multi-dimensional overlay of peers that is maintained by the CAN system [5], so the key points of CAN are summarized in the following. Each peer that connects to the network is assigned an index whose value is a point in a multi-dimensional space, randomly computed through a hash function. The peer will be responsible for a region of the space that contains that point. As the network evolves, the whole space is partitioned and fragmented into ever smaller regions, which are assigned to the joining peers.

Similarly, each resource published by a CAN participant is assigned a key in the same multi-dimensional space, using another hash function, and the key is consigned to the peer that is responsible for the region that includes the key. A discovery request issued to find this key, or any other key that belongs to the same region, is driven to this peer, exploiting the multi-dimensional overlay and the ordering of peer indexes along the different dimensions.

Some relevant properties of a CAN overlay unfolded over D dimensions are the following (see [5] for more details):

- the multi-dimensional space is toroidal, to avoid border effects; without losing generality, the values assigned to peer indexes are in the range (0,1) for each dimension;
- each peer is connected to $2D$ neighbor peers, i.e., to two adjacent peers per dimension;
- the search path follows the connections among adjacent peers. If the D-dimensional space is equally partitioned among N_p peers, the average path length is $(1/4) \cdot D \cdot N_p^{1/D}$;
- if the value of D is approximately logarithmic with respect to the number of peers N_p, other important properties are also kept logarithmic. For example, if $D \simeq (\log_2 N_p)/2$, the average length of the search path is of the order of $(\log_2 N_p)/2$ and the number of a peer's neighbors is of the order of $\log_2 N_p$.

Self-CAN uses the same overlay as CAN, but decouples resource keys from peer indexes. Each resource is assigned a multi-dimensional key via a hash function or, a useful alternative, through a semantic-aware mechanism. For example, the value of the key over each dimension may correspond to a property of the resource. The flexibility in the definition of the key space allows the efficient management of resource classes, a "class" being defined as the set of resources having specified common properties. For example, a class of resources could include the hosts having similar CPU and memory capabilities, or the software tools that offer a specified set of mathematical services. The resources of a class can be assigned the same value of the key, enabling the efficient execution of class and range queries, generally impossible in structured P2P systems.

The basic idea of Self-CAN is to let the resource keys self-organize along the D-dimensional space while preserving the sorting of key values along each dimension. This allows queries to discover a target key without knowing in advance the identity of the peer that is responsible for the key. As the performance evaluation section will confirm, the efficiency of discovery operations is unaltered with respect to CAN, but other important advantages are obtained, such as improved adaptivity, better load balance, and the possibility of associating a semantic meaning to the keys.

The keys are sorted with simple *pick* and *drop* operations by a multitude of ant-based mobile agents. Each peer computes its "centroid", which is the D-dimensional vector of real values that minimizes the distance from itself and the values of the keys stored in the local region of the network. The agents move and sort the keys using the peer centroid as a reference. To clarify this aspect, let us assume that

the key space is bi-dimensional and keys assume integer values in the range {0..15} for each dimension, which means that resources are categorized in 256 classes. Let us assume that the keys stored in a local region, which includes the local peer and the peers adjacent to it, are the following: (1,15), (3,14), (5,0) and (3,1). In this case the peer centroid is (3,15.5), since the value of the centroid for each dimension minimizes the average distance between itself and the values that the local keys assume on the same dimension[1]. The objective of the agents is to move the keys considering their relative distance with respect to the centroid. In the mentioned example, with the centroid equal to (3,15.5), the key (1,15) should be picked and moved towards a "predecessor" peer along the first dimension, whereas the key (3,1) should be moved towards a "successor" peer along the second dimension. A predecessor (successor) peer is the adjacent peer that, owing to the sorting process, is supposed to have a lower (higher) value of the centroid for the dimension of interest. An agent that arrives at the local peer carrying the key (3,15), previously picked on another peer, should drop the key, because its value is similar to the centroid. Pick and drop operations gradually order the keys over the D-dimensional structure, and then keep the ordering even when the environment changes, for example when peers disconnect and reconnect again, or when resource properties, and their keys, are modified. More details about agent operations are given in Section 3.

The Self-CAN prototype was used to test a simple scenario in which 16 peers are aligned along a 2-dimensional overlay, and each peer publishes on average 50 resources with key values in the range {0..15} for both dimensions[2]. The key values are assigned randomly, therefore the network starts from a disordered situation. The operations of 16 agents, generated one by each peer, rapidly reorder the keys. Figure 1 offers a graphical representation of the obtained ordering, by reporting the centroids of the peers at the end of the experiment. It can be noticed that centroid values are sorted along both dimensions, which means that the values of single keys are also sorted. Yet there is no predetermined association between key values and peers, as in CAN. A discovery message, issued to find a target key, is easily driven, following the gradient of centroid values, and hopping from peer to peer, towards the peer whose centroid is equal or as close as possible to the target key. As the performance section will show, the distribution of keys stored locally is centered on the value of the peer centroid and is very narrow, therefore all the keys with the desired value can be found in this peer or, at worst, in the adjacent peers. In the case of a range query, all the desired keys will be found in the neighbor peers, as discussed in Section 4.3.

3. AGENT OPERATIONS IN SELF-CAN

As will emerge from this section, the reordering process performed by agents is decentralized, self-organizing, and does not require any predefined association among resource keys and peer indexes. Agents are generated and die like

[1]For each dimension, key values are defined in a circular space, in which value 0 succeeds value 15: the distance between two values can be seen as the length of the minimum circle segment that separates these values.

[2]As discussed later, the intervals do not have to be equal: different ranges of values can be assigned to different key coordinates.

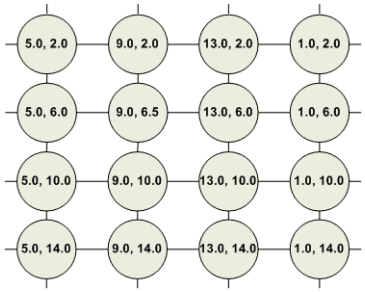

Figure 1: A 2-dimensional Self-CAN network with 16 peers. The values of centroids are reported for each peer. At the end of the experiment, they are sorted along both dimensions.

the real ants from which they are inspired. Each peer, at the time that it connects to the network, generates an agent with a probability P_{gen}. The agent lifetime is randomly generated with a statistical distribution whose average is equal to the average connection time of the connecting peer, calculated on the past activity of the peer. In this way, the average number of agents $\overline{N}_a$ that circulate in the network at a given instant of time is associated with the average number of peers connected in the network at the same time, $\overline{N}_p$,

$$\overline{N}_a \cong \overline{N}_p \cdot P_{gen} \tag{1}$$

Each Self-CAN agent, in its lifetime, performs a few simple operations, cyclically:

1. while it is not carrying any key, it hops from a peer to an adjacent one, chosen randomly;
2. at any new peer, it decides whether or not to *pick* a key from this peer;
3. while holding a key, the agent jumps to a new adjacent peer, trying to move towards a region in which the peer centroids are more similar to the value of the key;
4. at any new peer, the agent decides whether or not to *drop* the carried key.

Pick and drop operations are executed on the basis of Bernoulli trials whose probabilities depend on the similarity between the key k under evaluation and the centroid c of the local peer. This similarity, referred to as $f(k,c)$, is defined as:

$$f(k,c) = 1 - \frac{1}{D}\sum_{i=1}^{D}\frac{\Delta_i}{L_i/2} \tag{2}$$

where Δ_i is the distance between k and c evaluated along dimension i, and L_i is the length of the facet of the D-dimensional space of keys along dimension i, i.e., the number of values that a key may assume on the corresponding coordinate. In a toroidal space, $L_i/2$ is the maximum distance between two points along dimension i and is used in the fraction denominator to normalize the distances over the different dimensions. The second term in (2) can be considered as the normalized "Manhattan" distance between k and c. Indeed, it is the sum of the normalized distances along the different dimensions, divided by D. In this way, the value

of $f(k,c)$ ranges between 0 (minimum similarity) and 1 (k and c have exactly the same value).

The four steps of the agent life cycle are now described in detail. As long as the agent is unloaded, it randomly selects a dimension and a direction for its next movement (step 1). At any new peer (step 2), it evaluates the pick probability function, which for a local key k is defined as:

$$P_{pick}(k) = \frac{\alpha_p}{\alpha_p + f(k,c)} \quad \text{with } 0 \leq \alpha_p \leq 1 \qquad (3)$$

The expression for $P_{pick}(k)$ guarantees that the probability of picking a key k at a peer with centroid c is inversely proportional to the similarity between k and c. Therefore, the keys that are distant from the peer centroid are very likely to be picked, whereas the keys that are close to the centroid are picked with low probability because they are probably placed in the correct place. The parameter α_p can be tuned to modulate the pick probability. In fact, the probability is equal to 0.5 when the values of α_p and $f(k,c)$ are comparable, whereas it approaches 1 when $f(k,c)$ is much lower than α_p (i.e., when the key k is very different from the peer centroid) and 0 when $f(k,c)$ is much larger than α_p (i.e., when the key k is similar to the centroid). In this work, α_p is set to 0.1 as in the base ant algorithm introduced in [9].

At step 3, while carrying a key[3], the agent selects the dimension along which the normalized distance between the key and the local centroid is the largest. If the key is higher than the centroid on the selected dimension, the agent moves to the successor peer (i.e., the adjacent peer having a higher code on that dimension in the underlying CAN structure), otherwise it moves to the predecessor. In this way, the key ordering always respects the same direction as the ordering of peers established by the CAN structure.

At any new peer (step 4), the agent tries to drop the key with a Bernoulli trial with probability:

$$P_{drop}(k) = \frac{f(k,c)}{\alpha_d + f(k,c)} \quad \text{with } 0 \leq \alpha_d \leq 1 \qquad (4)$$

where k is the value of the carried key, c is the centroid of the new peer, and $f(k,c)$ is the similarity between the two values. If the drop operation is not performed, the agent continues its travel towards a region of the network where the key should be deposited, and retries the drop operation at every new peer. As opposed to $P_{pick}(k)$, $P_{drop}(k)$ grows with the similarity between k and c, therefore the agent tends to drop a key if it is similar to the other keys stored in the local region. The parameter α_d is set to a higher value than α_p, specifically to 0.5, in order to limit the frequency of drop operations, and help the agent move to an appropriate region where to drop the key[4].

Pick and drop operations contribute to the correct reordering of keys, because the agents tend to place every key in a peer that has a centroid value close to the key value. The progressive sorting is guaranteed by the fact that the centroid of a peer is calculated not only on the keys stored in the peer itself, but also on the keys stored by the adjacent peers along the D dimensions.

[3]To keep the key available while it is carried by an agent, a simple redundancy mechanism is adopted: the peer from which the key has been taken maintains a copy and discards it only when alerted by the peer where the key is successively dropped.

[4]It should be remarked here that the values of α_p and α_d only affect the rapidity of the reordering process when starting from a chaotic condition, but their setting have a very small effect on the operations in normal conditions, when the ordering must be maintained and refined.

4. PERFORMANCE ANALYSIS OF SELF-CAN

To analyze the behavior of Self-CAN in large networks, a set of experiments were performed with a Java event-based simulator. Java objects are used to model the peers and the mobile agents that perform the operations described in Section 3. The simulator, as well as the prototype, is available at http://self-can.icar.cnr.it.

It is assumed that the average number of resources published by a peer is extracted from a Gamma probability distribution, with average equal to 15, unless otherwise stated. The key value of each published resource is uniformly distributed within the range of admissible values for each coordinate of the key. Therefore at the beginning key values are distributed randomly in the network; afterwards, the keys are sorted through the operations of Self-CAN agents. The P_{gen} probability is set to 1.0, which means that each new or reconnecting peer issues one mobile agent. The peer churning is modeled as follows. When a peer connects, its connection time is decided according to a Gamma distribution with the mean value that is typical of the peer. When the connection time expires the peer disconnects, and after a time interval generated in the same fashion, it reconnects to the network. The average connection time for all the peers, T_{peer}, is set to 5 hours. The lifetime of an agent is set to the average connection time of the peer that generates the agent. After receiving an agent, a peer forwards it to the next peer after a random interval T_{mov}. Since the Self-CAN procedures can be accelerated or decelerated by tuning the value of T_{mov}, this parameter will be used as a time unit and the performance results versus time will be reported accordingly.

Experiments were performed to evaluate different aspects: the capacity of the algorithm to reorder the resource keys over the multi-dimensional space, the efficiency and effectiveness of resource discovery operations, for both punctual and range queries, the storage and traffic load, and the dynamic behavior. All these aspects are described in the rest of this section.

4.1 Analysis of the Reordering Process

The reordering of keys can be considered successful if: (i) the peer centroids are sorted and uniformly spaced along the different dimensions, which means that the keys are also sorted, and (ii) the keys are clustered, i.e., in each peer they are similar to each other.

An efficient method to evaluate the first characteristic is to compute the average Manhattan distance, in the space of resource keys, between two "consecutive" centroids, i.e., the centroids of two adjacent peers. In a perfectly ordered network, if L_i is the number of distinct values that can be assigned to the coordinate i of a key, and N_i is the number of peers that cover the corresponding dimension of the peer space, the average Manhattan distance between the centroid values of two peers that are adjacent along dimension i must be comparable to L_i/N_i[5].

[5]If two peers are adjacent along dimension i, the distance on the other dimensions is null or very small.

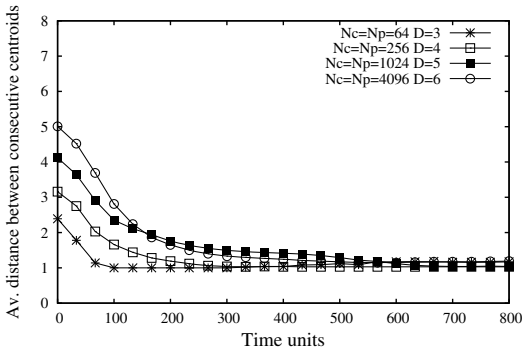

Figure 2: Average Manhattan distance between consecutive centroids in networks with different sizes.

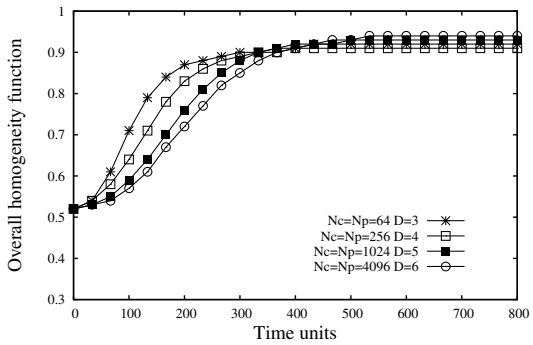

Figure 3: Overall homogeneity function in networks with different sizes.

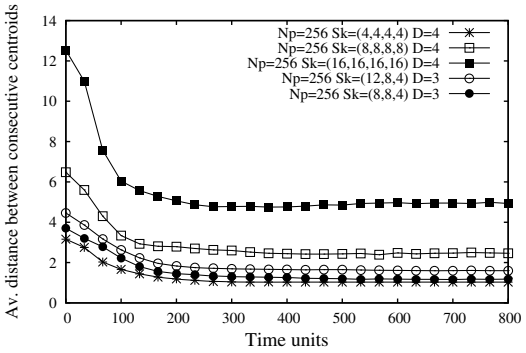

Figure 4: Average Manhattan distance between consecutive centroids in a network with fixed size and different settings for the vector S_k.

Let us consider the simple case in which the space of keys is a hyper-cube, i.e., the number of admissible key values is the same for each dimension, and the number of peers N_p is equal to the number of resource classes N_c. Each class is associated with a specific value of the multi-dimensional key. In this case, the expected Manhattan distance between two consecutive centroids, regardless the dimension on which the peers are adjacent, should be comparable to $\sqrt[D]{N_c}/\sqrt[D]{N_p}$=1. Figure 2 shows the trend of the centroid distance in four experiments in which the values of N_c and N_p are varied from 64 to 4096, and the number of dimensions D is equal to $(\log_2 N_c)/2$. In these experiments, each coordinate of the key is generated randomly in the range between 0 and $\sqrt[D]{N_c} - 1$. At time 0, the sorting algorithm is started. The figure shows that the average value of the centroid distance rapidly converges to the expected value, confirming that the centroids are reordered correctly. The time to convergence increases with the number of peers, but convergence is reached at between 300 and 500 time units in all the considered cases. If, for example, the time T_{mov} is 5 s, this corresponds to a time between 25 and 40 minutes. Notice that these experiments are performed starting from a chaotic situation, in which the keys and the centroids are completely disordered. In a real situation, the peers join and leave the network gradually, and the publishing/removal of resources is also gradual: the correct and gradual placement of a relatively small number of new keys, in a network that is already ordered, is a much easier and faster task. This issue will be discussed in Section 4.5.

The clustering property is assessed by verifying whether the keys placed on a peer are similar to each other. To this aim, the homogeneity function of a peer, H_p, is defined as:

$$H_p = 1 - \frac{\sum_{(k_x,k_y)} d(k_x, k_y)}{n_k} \qquad (5)$$

where $d(k_x, k_y)$ is the normalized Manhattan distance between two keys, k_x and k_y, which are stored by peer p, and n_k is the number of such couples. The homogeneity function of a peer can assume values between 0 and 1, and higher values correspond to high degrees of clustering in the peer. The overall homogeneity function, H, is defined as the value of H_p averaged over all the peers. In a disordered network, with randomly distributed resources, the homogeneity function is equal to about 0.5. As the keys are reordered and clustered, the value of H should become increasingly higher. This is confirmed by Figure 3, which shows the overall homogeneity function computed during the experiments described before. The value of H, after a rapid increase in the transient phase, stabilizes to a value higher than 0.90. It can be noticed that the index is hardly affected by the network size, which confirms the scalability properties of Self-CAN.

As mentioned before, in Self-CAN the keys can be defined in a flexible way, and the range and number of admissible values can be different for each dimension. To test this aspect, let keys range be $0...L_i - 1$ for dimension i, with $i = 1...D$, and define the key space size through the vector S_k=$(L_1, L_2, .., L_D)$. The overall number of classes N_c is, thus, equal to $\prod_1^D L_i$. In the case that the L_i values are not all equal, the space of keys is not a hyper-cube, but a D-dimensional hyper-rectangle.

Figure 4 shows the average Manhattan distance between consecutive centroids in a network with fixed size, 256 peers, and different settings for S_k. In the first three experiments, the number of dimensions is set to 4, and the number of admissible attribute values is set to 4, 8 and 16 for each dimension. The average centroid distance converges to values slightly higher than the minimum possible values, which are, respectively, 1, 2 and 4. In the other two experiments, the space of keys is a 3-dimensional hyper-rectangle: also in these cases the ants sort the keys rapidly.

The reason why the average centroid distance is slightly higher than the minimum possible value is related to the statistical nature of the process. This can be seen by reexamining Figure 1 (16 peers organized in 2 dimensions and S_k=$(16, 16)$): in the second peer of the second row the y-

component of the centroid is not perfectly spaced with the others: its value is 6.5 while it should be 6.0 in a perfect ordering. The average Manhattan distance between this peer and the four adjacent ones, considering them in clockwise direction starting from the left, is $(4.5 + 4.5 + 4.5 + 3.5)/4 = 4.25$. In turn, the average distances between the four adjacent peers and their respective neighbors are $(4 + 4 + 4.5 + 4)/4 = 4.125$, $(4 + 4 + 4 + 4.5)/4 = 4.125$, $(4.5 + 4 + 4 + 4)/4 = 4.125$, and $(4 + 3.5 + 4 + 4)/4 = 3.875$. For all the other peers of the network the average Manhattan distance is 4. It turns out that the average Manhattan distance between two adjacent peers is slightly higher than 4, the minimum possible value in this case. Despite these small imperfections, the sorting of centroids is sufficiently accurate, which ensures that discovery procedures can be executed efficiently, as the next section will show.

4.2 Performance of the Discovery Procedure

The purpose of a discovery procedure is to find the resources belonging to a specific class, i.e., which have been associated with a specific value of the multi-dimensional key. This is a typical problem in Grid environments, where a user needs to locate a number of resources that address his/her requirements, for example a set of hosts having specific CPU and memory capabilities.

Before analyzing the performance of the discovery process, it must be verified that the keys with a specified value can be retrieved in the peers whose centroids have equal or similar values. In this is true, the important consequence is that a search for a target key can be converted in a search for a peer centroid. Figure 5 shows the histogram of the Manhattan distance between a key and the local centroid, evaluated over all the keys of a Self-CAN network with 256 peers, 256 resource classes, and key pattern S_k=(4, 4, 4, 4). The figure reports the histogram observed before the start of the sorting process and in a steady condition. Notice that whenever the centroid coordinates are integer values, as it is usually the case, the distance is integer also. Occasionally, the centroids have fractional values over some dimension and, correspondingly, the keys have fractional distances from them; since this occurs rarely, the histogram has low values for fractional distances. Before the process starts, when the keys are placed randomly, the average Manhattan distance between two keys is 4, since the average distance along each dimension is 1, i.e., one forth of the facet length measured in the key space. In fact, the distribution of the distances between key and centroid (top plot of Figure 5) is centered on a value slightly lower than 4, and has a typical Gaussian shape. In a steady condition (bottom plot of the same figure), about 60% of the keys are exactly equal to the local centroid, and almost all the remaining keys have a distance from the centroid equal to 1. This means that the coordinates of the key and the centroid are equal on at least three dimensions, and may only differ by 1 along a single dimension. The percentage of Manhattan distances higher than 1 is negligible. Therefore, a search process can find about 60% of the keys having a specified value in the peer whose centroid is equal or very similar to the key, and the remaining keys in the $2D$ adjacent peers.

The discovery algorithm is very simple. At every step, the peer that receives the search message (or, at the first step, the peer that generates the request) computes the normalized distances, along the different dimensions, between

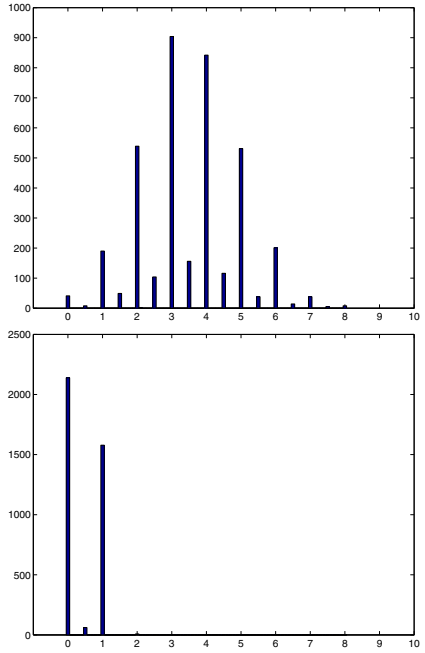

Figure 5: Histogram of the Manhattan distance between a key and the local centroid: before the start of the sorting process (top plot), and in a steady condition (bottom plot).

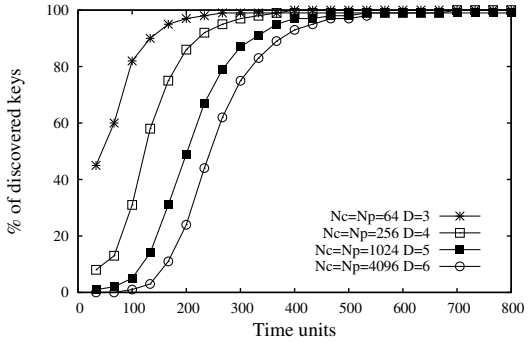

Figure 6: Percentage of discovered keys in networks with different sizes.

the centroid of the local peer and the target key. The peer evaluates the largest of these distances, and forwards the message along the corresponding dimension, to the successor or predecessor peer, depending on the value of the key being larger or lower than the centroid on this dimension. This corresponds to following the gradient of centroid values towards the peer whose centroid is the most similar to the target key. Once the search path terminates - because it is no longer possible to decrease the distance between the target key and the peer centroid - the target keys are collected in the local peer and in the $2D$ adjacent peers, and are delivered to the peer that originated the request.

Figure 6 shows the average percentage of discovered keys with respect to the overall number of keys that have the target value; keys are searched for while they are being ordered starting from a chaotic initial distribution, as in the cases discussed above. In these experiments, the key space

is a hyper-cube, and the number of admissible values of keys is 4 for each dimension. The figure confirms that discovery procedures find practically all the keys, once they have been ordered by the ant-based process.

For the same experiments, the average path length is reported in Figure 7, along with the 1st and 99th percentiles. The average number of hops of search messages is comparable to $(\log_2 N_p)/2$; as discussed in the introductory section, this is the same value obtained using the basic CAN system. Therefore, Self-CAN preserves this fundamental property of CAN, despite not being obtained with a predefined association between keys and hosts, but through the self-organizational behavior of the ant algorithm. Moreover, the 99th percentile is always comparable to $\log_2 N_p$, which means that the logarithmic behavior is ensured also in the most unfortunate cases.

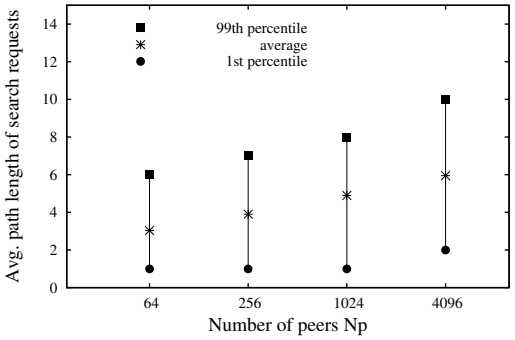

Figure 7: Average, 1st and 99th percentile of the path length of discovery procedures in networks with different size.

4.3 Performance of Range Queries

The previous section focused on the performance of punctual queries, issued to discover the keys having a given value. Self-CAN can also efficiently manage range queries, thanks to the sorting of keys over the multi-dimensional space. The number of desired resources and the time and cost that the user can afford for the research can be different in different contexts. Accordingly, two approaches were devised to serve range queries: the *sweep up* approach and the *explosion* approach.

To illustrate the two techniques, let us consider the case in which the set of target keys is defined by a closed interval over two dimensions. With the *sweep up* approach, illustrated in top part of Figure 8, the first objective is to drive the query message from the generating peer (marked as P in the figure) towards the closest vertex of the two-dimensional region defined by the range query. This region, highlighted in the figure, includes the peers whose centroids are within the range intervals of the query. Then, a message is forwarded along one of the borders of the region, in this case the lower horizontal border. In turn, every border peer reached by the message forwards the query along the vertical dimension, up to the upper horizontal border. The target keys are collected by all these messages along their path; as a message reaches the border of the target region, it is directly forwarded to the peer P. The generalization of this technique to target regions defined on more than two dimensions is straightforward and is not discussed

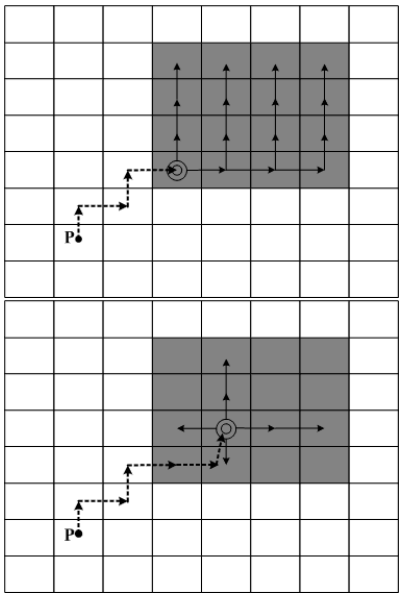

Figure 8: Path of range queries with the two defined strategies: sweep up (top), explosion (bottom).

further. The objective of this approach is to explore all the peers that are located in the target region and retrieve as many desired keys as possible. Conversely, the goal of the *explosion* approach is to collect a consistent number of target keys by contacting a much lower number of peers. This time, the query message is driven to a peer that is located in the core of the target region. Then, two query messages are forwarded along each dimension of the range region, in the two opposite directions, up to the border [6]. The query path is depicted in bottom part of Figure 8. Of course, the number of messages is lower, as well as the number of contacted peers, but it is no longer possible to collect all the target keys.

To compare the two approaches, a set of experiments are performed in a network with D=5, key pattern S_k=(4, 4, 4, 4, 4), and 1024 peers. The range of target keys is defined by a vector R_k, whose $i-$th element specifies the size of the range of values that are searched for in dimension i. Two cases are considered. In the first case, R_k=(3, 3, 1, 1, 1) specifies a range of size 3 (the key can have any of three contiguous values) over the first two dimensions and a specific individual value over the remaining dimensions. In the second case, R_k=(3, 3, 3, 1, 1) means that the target region is extended in the third dimension. Figure 9 shows the average percentage and number of discovered keys, and the average number of contacted peers. The sweep up approach finds all the keys in the defined range, but the number of contacted peers is considerably higher. Conversely, the explosion approach finds a fraction of the target keys by examining a smaller number of peers. The difference between the two approaches increases

[6]A number of target keys can be stored by the peers that are located just outside the target region, and are adjacent to border peers. Since peers that are adjacent to each other can easily advertise their respective keys, it is convenient to forward the search messages to these "external" peers if they are known to possess target keys. This strategy is used in both the examined approaches.

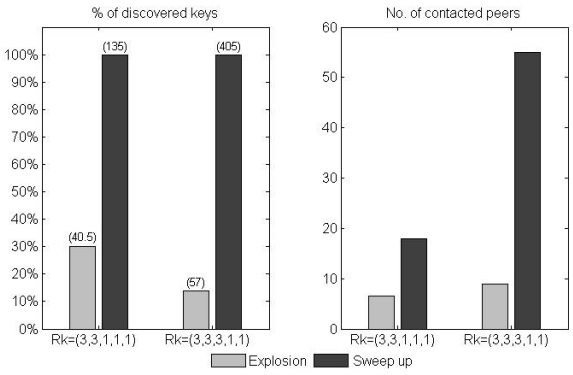

Figure 9: Percentage of discovered keys (the absolute number is also indicated on top of the bars) and number of contacted hosts with sweep up and explosion approach, for two types of range queries. The network has 1024 peers and the key pattern is S_k=(4,4,4,4,4).

with the volume of the target region. Indeed, with the sweep up approach, the number of contacted peers is of the order of the product of the elements of R_k, while it is simply of the order of the sum of the elements of R_k with the explosion approach. In conclusion, the appropriate approach should be chosen depending on the application requirements: for example, in an OLAP analysis the user may want to find all the target keys, while s/he could accept finding several results (but not all) if the goal is to find a set of Grid hosts with given requirements.

As a conclusive remark, both strategies are feasible because the ordered keys are allowed to have semantic values, associated with resource attributes. In classical structured P2P systems, the key values are spread by hash functions, therefore range queries can only be served by issuing as many queries as the punctual key values included in the target range, or by using additional structures. For more comments on this issue, please see the related work section.

4.4 Load Balancing

An important characteristics of Self-CAN is its capability of fairly distributing the load among the peers. In this respect, Self-CAN has two important advantages when compared to CAN:

1. In CAN the number of keys stored by a peer is proportional to the volume of the zone assigned to the peer; for this reason, CAN introduces a "uniform partitioning" technique to balance the zones assigned to peers. In Self-CAN, the multi-dimensional structure is not used to assign keys to peers, but as a substrate that allows ant agents to order the keys. Therefore, the volume of the zone assigned to a peer has no effect on the number of keys that the peer stores, and there is no need to devise a technique for uniform partitioning.

2. Even with the use of improved partitioning strategies, CAN cannot guarantee a true load balancing in the case that some keys are more popular than others: these keys put a higher load on the peers that host them. In Self-CAN, the load distribution among the peers is not affected by the popularity distribution of keys: pick and drop operations distribute the keys to the peers in a fair fashion. It results that more popular keys may be distributed among a larger number of contiguous peers, but the average load of a single peer is stable.

20/0	20/0	20/0	20/0	20/0	20/0	20/0	20/0
20/0	20/0	20/0	20/0	20/0	20/0	20/0	20/0
20/0	20/0	20/0	20/0	20/0	20/0	20/0	20/0
20/0	20/0	20/0	**50/50**	**50/50**	20/0	20/0	20/0
20/0	20/0	20/0	**50/50**	**50/50**	20/0	20/0	20/0
20/0	20/0	20/0	20/0	20/0	20/0	20/0	20/0
20/0	20/0	20/0	20/0	20/0	20/0	20/0	20/0
20/0	20/0	20/0	20/0	20/0	20/0	20/0	20/0

18/0	18/0	19/0	18/0	20/0	17/0	22/0	21/0
20/0	19/0	20/0	22/0	19/0	20/0	18/0	18/0
19/0	19/0	23/0	**24/10**	**23/11**	21/0	20/0	23/0
25/0	21/0	**26/13**	**27/27**	**28/28**	**26/8**	25/0	22/0
25/0	21/0	**26/7**	**27/27**	**28/28**	**28/8**	23/0	23/0
21/0	24/0	19/0	**26/16**	**25/17**	20/0	20/0	19/0
18/0	23/0	19/0	24/0	24/0	18/0	23/0	22/0
20/0	19/0	24/0	25/0	24/0	21/0	21/0	17/0

Figure 10: Distribution of keys for the load balancing test, at the beginning of the process (top) and in steady situation (bottom).

To illustrate this, we set up a specific experiment using the prototype: in a network with D=2, N_p=64, and key pattern S_k=(8, 8), the centroids are assumed to be uniformly spaced, with values from 0 to 7 for both dimensions, and all the keys are the same as the local centroid. All the peers store 20 keys except the four central peers, which store 50 keys. This scenario is described in the top part of Figure 10. The label N/M on each peer means that N keys are stored by the peer, of which M belong to the four most "popular" classes, which are (3,3), (3,4), (4,3) and (4.4). For example, the first peer of the first column has centroid (0,0), and stores 20 keys with the same value. The forth peer of the forth column has centroid (3,3) and stores 50 such keys.

The algorithm is started in this unbalanced situation and, after about 500 time units, the system gets to a steady situation. A snapshot, taken at this point from the prototype, and depicted in the bottom part of Figure 10, shows that popular keys have been diffused from central peers to their neighbors, and load balance is much fairer than at the beginning. This new equilibrium is the result of two phenomena: on the one hand, keys tend to diffuse out of heavily loaded peer, because agents perform pick Bernoulli trials on a larger number of keys; on the other hand, pick and drop operations tend to keep similar keys close to each other. The first phenomenon improves load balancing, while the second facilitates resource discovery operations, because target keys are better clustered. Interestingly, we found that this equilibrium can be biased towards one behavior or the other by tuning the parameters α_p and α_d of pick and drop proba-

bility functions. This study is not reported here for lack of space.

4.5 Processing Load and Dynamic Behavior

Self-CAN improves CAN also in terms of network and processing load. In a structured system like CAN, the keys of new resources, for example those published by new or reconnecting peers, must be immediately placed in specified hosts: this can originate a high load if many resources are published in a short interval of time. In Self-CAN, the load is invariant because a new peer does not need to perform any additional operation (besides the operations related to the overlay management, such as the update of the list of neighbors): the keys of the new resources will be picked by the agents that pass by this peer. The processing load Λ can be defined as the average number of agents per second that arrive and are processed at a peer. Λ can be computed by multiplying the average number of agents $\overline{N}_a$ by the frequency of their movements $1/T_{mov}$, so obtaining the number of times per second that an agent arrives at any peer, and then dividing the result by the average number of peers $\overline{N}_p$ to get the number of times per second that an agent arrives at a single peer,

$$\Lambda = \frac{\overline{N}_a}{\overline{N}_p \cdot T_{mov}} \approx \frac{P_{gen}}{T_{mov}} \quad (6)$$

The simplification is given by applying expression (1) of Section 2.

For example, if the average value of T_{mov} is equal to 5 seconds, and $P_{gen} = 1.0$, each peer receives and processes about one agent every 5 seconds, which is an acceptable load, since pick and drop operations are very simple. Note that the processing load does not depend on the frequency of peer joinings and disconnections nor on the network size, which confirms the scalability properties of Self-CAN.

The results discussed in Sections 4.1 and 4.2 showed that the Self-CAN agents can reorder the keys starting from a completely disordered network. Normal circumstances are much less stressful: if the network grows gradually, the correct sorting of the keys can be maintained with a few agent operations. In particular, the placement of a new key in an ordered network can be performed in logarithmic time, since it corresponds to the discovery of the peer centroid that is the closest to the key value. In a number of experiments performed in the same scenarios as those considered in Sections 4.1 and 4.2, a new peer joined the network at time unit 600, when ordering of keys can be considered stable, and published 15 resources with randomly chosen keys. After the arrival and the pick/drop operations of 20 to 25 agents, the centroid and the keys of the new peer were perfectly ordered.

The disconnection of a peer is also simple to manage: if the peer leaves the network gracefully, the keys are passed to the adjacent peers, and moved by agents if necessary. To handle the abrupt disconnection of a peer, a mechanism is necessary, based on some redundancy and periodical soft-state updates among adjacent peers. Of course, a mechanism of this kind is necessary in any P2P system, it is not a peculiar requirement of Self-CAN.

Finally, Self-CAN is robust with respect to changes of resource properties: if the value of a key is modified, the key is quickly moved by the agents that, by recognizing that the key has become an outlier in the current peer, assign a large pick probability to it.

5. RELATED WORK

The P2P paradigm is increasingly adopted as a valuable alternative to centralized and hierarchical architectures for the management of large-scale computing systems. The fundamental characteristics that should be provided by efficient and versatile information systems have been individuated by the ICT community [11, 12] as: self-organization (meaning that components are autonomous and do not rely on any external supervisor), decentralization (decisions are to be taken only on the basis of local information) and adaptivity (mechanisms must be provided to cope with the dynamic characteristics of hosts and resources).

Recently, swarm and bio-inspired algorithms proved capable of considerably improving the performance of P2P computer systems [7]. Two interesting examples are Anthill and BlatAnt. The Anthill project [13] is tailored to the design, implementation and evaluation of P2P applications based on multi-agent and evolutionary programming. The devised system is composed of a collection of interconnected *nests*. Each nest is a peer entity that makes its storage and computational resources available to swarms of *ants*, mobile agents that travel the network to satisfy user requests. BlatAnt is an ant-inspired algorithm that creates P2P overlay networks with bounded diameters [14]. Ant-inspired agents are used to rewire connections among nodes, which helps to limit the path length of search messages. Both Anthill and BlatAnt system are unstructured: this implies that discovery procedures are fundamentally "blind", and can be inefficient in terms of traffic load and response time, even if caching mechanisms may help to increase their performance. In Self-Chord [10], ant algorithms proved capable of triggering a self-organization behavior also in ring-structured P2P systems.

As opposed to the mentioned systems, Self-CAN is able to efficiently support complex and range queries on multiple attributes. This is indeed a very tough issue in P2P systems [15], and particularly in structured ones, because the use of DHT techniques tends to disperse the keys of similar resources into distant places of the structure. Some types of structured systems are capable of serving range queries, but at the cost of either maintaining complex auxiliary structures, such as tree or trie overlays [16], or increasing the traffic load by issuing a number of sub-queries [17]. The Squid discovery protocol [18] uses a dimension-reducing technique, the space filling curve (SFC), to map multi-attribute keywords to a mono-dimensional space, i.e., a ring structure similar to that used by Chord. This enables Squid to support queries defined through partial keywords, wildcards and ranges. However, SFCs have an important drawback, in that a region in the multi-attribute space can be mapped to different and distant segments on the ring. In the case of a range query, a system like Squid must then forward separate query messages to all these segments, which of course may notably increase the response time and the network load.

Mercury [19], like Self-CAN, avoids the use of hash functions to compute key values, and supports multi-attribute range queries. Mercury maintains a *routing hub* - a logical collection of nodes - for each attribute. Each node within a hub is responsible for a range of values of the particular attribute. A range query is served by first trying to determine the most selective attribute of the query, and then driving the query through the corresponding logical hub. However,

Mercury does not scale well with the number of attributes, because (i) a key must be replicated and inserted in as many hubs as the attributes for which the key is given a value, and (ii) in addition to intra-hub links, inter-hub links are also necessary to forward the query to the desired routing hub. As opposed to Mercury, Self-CAN does not need any additional structure, like logical hubs, thus preserving the simplicity, efficiency and flexibility of the basic CAN overlay.

6. CONCLUSION

This paper presents Self-CAN, a self-organizing P2P system. Self-CAN inherits from CAN the regular multi-dimensional structure according to which peers are organized; the distribution of resource keys to the peers is, instead, performed by ant-inspired agents that travel in this regular structure moving keys through very simple "pick" and "drop" operations. It results that keys are not statically assigned to peers, as they are in CAN, but are dynamically sorted and distributed to peers. This makes Self-CAN robust to peers' churning, adaptable to heterogeneous and dynamic popularity of the keys, fair in load balance. Finally, and most notably, by decoupling the space of the peers from the one of the keys, Self-CAN allows a semantic meaning to be associated with the coordinates of the keys, so that each dimension of the key space can be used for a different attribute of the resources. In this fashion, complex range queries can be solved easily and efficiently, which makes Self-CAN particularly well suited to Grid environments.

7. REFERENCES

[1] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon highly available key-value store," Amazon, Tech. Rep. http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf, October 2007.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. of the Conference on Applications, technologies, architectures, and protocols for computer communications SIGCOMM'01*, 2001.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2001, pp. 161–172.

[6] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," *Lecture Notes in Computer Science*, vol. 2218, pp. 329–350, 2001.

[7] S. Y. Ko, I. Gupta, and Y. Jo, "A new class of nature-inspired algorithms for self-adaptive peer-to-peer computing," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 3, no. 3, pp. 1–34, 2008.

[8] A. Forestiero and C. Mastroianni, "A swarm algorithm for a self-structured P2P information system," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 4, pp. 681–694, August 2009.

[9] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm intelligence: from natural to artificial systems*. New York, NY, USA: Oxford University Press, 1999.

[10] A. Forestiero, E. Leonardi, C. Mastroianni, and M. Meo, "Self-chord: a bio-inspired p2p framework for self-organizing distributed systems," *IEEE/ACM Transactions on Networking*, vol. 18, no. 5, pp. 1651–1664, October 2010.

[11] A. Iamnitchi and I. Foster, "A peer-to-peer approach to resource location in grid environments." Norwell, MA, USA: Kluwer Academic Publishers, 2004, pp. 413–429.

[12] I. J. Taylor, *From P2P to Web Services and Grids: Peers in a Client/Server World*. Springer, 2004.

[13] O. Babaoglu, H. Meling, and A. Montresor, "Anthill: A framework for the development of agent-based peer-to-peer systems," in *Proc. of the 22 nd International Conference on Distributed Computing Systems ICDCS'02*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 15–22.

[14] A. Brocco, F. Frapolli, and B. Hirsbrunner, "Blåtant: Bounding networks' diameter with a collaborative distributed algorithm," in *Proc. of the 6th InternationalConference on Ant Colony Optimization and Swarm Intelligence, ANTS 2008*, Brussels, Belgium, 2008, pp. 275–282.

[15] A. S. Cheema, M. Muhammad, and I. Gupta, "Peer-to-peer discovery of computational resources for grid applications," in *Proc. of the 6th IEEE/ACM International Workshop on Grid Computing*, Seattle, WA, USA, November 2005, pp. 179–185.

[16] J. Albrecht, D. Oppenheimer, A. Vahdat, and D. A. Patterson, "Design and implementation trade-offs for wide-area resource discovery," *ACM Transactions on Internet Technology*, vol. 8, no. 4, pp. 1–44, 2008.

[17] A. Andrzejak and Z. Xu, "Scalable, efficient range queries for grid information services," in *Proc. of the Second IEEE International Conference on Peer-to-Peer Computing P2P'02*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 33–40.

[18] C. Schmidt and M. Parashar, "Enabling flexible queries with guarantees in p2p systems," *IEEE Internet Computing*, vol. 8, no. 3, pp. 19–26, 2004.

[19] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 353–366, 2004.