Bio-Inspired P2P Systems: The Case of Multidimensional Overlay

RAFFAELE GIORDANELLI and CARLO MASTROIANNI, ICAR-CNR and eco4cloud srl, Italy
MICHELA MEO, Politecnico di Torino, Italy

This article presents an ant-based approach that enhances the flexibility, robustness and load balancing characteristics of structured P2P systems. Most notably, the approach allows peer indexes and resource keys to be defined on different and independent spaces, so that it overcomes the main limitation of standard structured P2P systems, that is, the need to assign each key to a peer having a specified index. This helps to improve load balancing, especially when the popularity distribution of resource keys is nonuniform, and enables the efficient execution of complex and range queries, which are essential in important types of distributed systems, for example, in Grids and Clouds. Beyond describing the general approach, this article focuses on the specific case of Self-CAN, a self-organizing P2P system that, while relying on the multidimensional 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. This system is particularly useful for the management and discovery of the resources that can be conveniently characterized by the values of several independent attributes.

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

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Bio-inspired, peer-to-peer, resource discovery, self-organizing

ACM Reference Format:
Giordanelli, R., Mastroianni, C., and Meo, M. 2012. Bio-inspired P2P systems: The case of multidimensional overlay. ACM Trans. Auton. Adapt. Syst. 7, 4, Article 35 (December 2012), 28 pages.
DOI = 10.1145/2382570.2382571 http://doi.acm.org/10.1145/2382570.2382571

1. INTRODUCTION

In recent years, peer-to-peer systems have definitely overstepped their mere role of middleware solution for traditional file-sharing applications and are now adopted in all kinds of large-scale distributed computing systems, thanks to their advantages over centralized and hierarchical solutions, among which their scalability and robustness properties.

New fields of application for P2P systems are Grid and Cloud Computing. In Grids [Foster and Kesselman 2003], different types of computing resources are 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

This article is an extended and enhanced version of the paper presented at the 8th IEEE/ACM International Conference on Autonomic Computing (ICAC 2011).
This research work was partially funded from the MIUR project FRAME, PON01_02477.
Authors' addresses: R. Giordanelli, eco4cloud srl, Piazza Vermicelli, Rende (CS), Italy; email: giordanelli@eco4cloud.com; C. Mastroianni, ICAR-CNR, Via P. Bucci 41C, Rende (CS), Italy; email: mastroianni@icar.cnr.it; M. Meo, Politecnico di Torino, Corso Duca degli Abruzzi, Torino, Italy; email: michela.meo@polito.it.

DOI 10.1145/2382570.2382571 http://doi.acm.org/10.1145/2382570.2382571

through their names, as in file sharing systems, but through a set of characteristics, which are described by resource metadata documents. P2P solutions can also be exploited to manage the information systems of big companies, like Google, Amazon, and Microsoft, which offer pay-as-you-go-services through their Cloud platforms [Armbrust et al. 2010]. For example, Amazon Dynamo [De Candia et al. 2007], a key-value storage system adopted by several Amazon's core services, can be considered as a variant of the Chord P2P system. Moreover, the research community is investigating the possible benefits of using the P2P paradigm in Clouds composed of multi-owner data centers. The so-called "federated Clouds" or "P2P Clouds" can be the most efficient solution for the provisioning of a wide range of resources at low or zero cost, and for the execution of applications where the physical location of nodes in important [Panzieri et al. 2011].

Most modern P2P architectures are "structured", meaning that peers are organized in a predefined structure, for example, a ring (as in Chord [Stoica et al. 2001]), a multidimensional grid (adopted by CAN [Ratnasamy et al. 2001]), a tree (as in Pastry [Rowstron and Druschel 2001]), or other structures that in most cases can be seen as an evolution or a combination of these basic three. A hash function is used to give each resource a key, and the key is assigned to a node whose code, also computed with a hash function, is equal or as close as possible to the key.

The main reason why structured P2P systems win against unstructured counterparts–for which the network evolves randomly and there is no predetermined way of assigning resources to peers–is that they use "informed" algorithms to drive user queries towards the desired keys in a short and bounded time [Androutsellis-Theotokis and Spinellis 2004]. Unfortunately, structured systems also share a major drawback, which stands in the fact that the discovery process is exclusively driven by the key value, and cannot take advantage from the knowledge of specific characteristics of the target resource [Rodrigues and Druschel 2010]. As a consequence, there is no simple way of efficiently serving *range* queries that need to discover a set of resources sharing common features, for example a set of machines with CPU speed and RAM memory comprised in a given range, or a set of mathematical software tools with given characteristics and whose cost does not exceed a specified value. The difficulty derives from the fact that the keys of similar resources are spread over the P2P overlay by the hash function. Range queries are particularly important both in Grid environments and in large Cloud infrastructures: for example, the manager of a Cloud data center often needs to rapidly individuate a set of physical machines whose CPU and RAM match the requirements of a Virtual Machine. Other notable drawbacks of structured P2P systems concern the load balance (the nodes that are assigned the most popular keys may be significantly more loaded than the others) and the dynamic behavior (e.g., 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 P2P networks by imitating the self-organizing behavior of biological systems, such as flocks of birds, insect swarms, and, above all, ant colonies [Ko et al. 2008]. 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 [Bonabeau et al. 1999]. These P2P systems are sometimes referred to as "self-structured" [Brocco et al. 2010; Forestiero and Mastroianni 2009], because their structure is constructed with self-organizing techniques.

In Self-Chord [Forestiero et al. 2010], an ant-inspired algorithm is used to sort resource keys over a Chord-like ring structure. The algorithm decouples resource keys from peer indexes, which allows key values to assume a semantic meaning, as they are no longer the result of a hash function but may be associated with the value of the

main resource attribute. This strategy facilitates the execution of range queries, as the keys of similar resources are stored by neighbor peers, and helps to improve load balancing and adaptivity. These advantages, though, are limited to the situations in which a single attribute can be used to characterize the resources, which is often a too-strict constraint. However, the ant-inspired technique can be generalized and applied to any kind of structure. Depending on the application domain, the appropriate type of overlay (ring, grid, tree, etc.) should be chosen to better match the nature of the resources and efficiently serve user queries.

This article has a twofold goal. On the one hand, it describes the key characteristics of the generalized ant-inspired strategy and gives hints on how it can be adapted to any particular overlay, and specifically to a ring, a multidimensional grid, and a tree. In this respect, this article follows the research avenue, discussed for example in Jelasity et al. [2009], which investigates how flexible and configurable protocols can be developed to tackle the plethora of overlay networks that have been proposed so far. On the other hand, this article focuses on the case of a multidimensional structure, specifically the one offered by CAN, extending the work presented in Giordanelli et al. [2011]. The analyzed system, Self-CAN, is particularly efficient for the execution of complex and range queries in the case that resources are characterized and indexed through a set of independent features, which is a very common situation. Performance evaluation was carried with both a Java prototype[1] and an event-based simulator, depending on the size of the analyzed network.

This article is organized as follows: Section 2 describes the generalized ant-inspired approach and gives details on how the approach can be adapted to three different types of overlay; Section 3 describes the Self-CAN model, specifically the operations performed by the ant-inspired agents and the discovery procedure; the performance of Self-CAN, with regard to its capacity of sorting the keys and serving user queries, is analyzed in Section 4; finally, after the related work section, Section 6 concludes the article.

2. A GENERALIZED ANT-INSPIRED STRATEGY FOR STRUCTURED P2P SYSTEMS

In any kind of structured P2P system, starting from the pathfinders, for example, Chord and CAN, to recent commercial systems, like Kademlia [Maymounkov and Mazières 2002], the main objective is to provide clients a way to rapidly discover the desired resources along the distributed overlay. This is done exploiting the "Distributed Hash Table" paradigm: every resource is given a key using a hash function applied to the resource name, and the key is assigned to a node of the structure whose code is equal or very close to the resource key. The hash function is needed to fairly distribute the resources over the structure, but this is possible only if the popularity of the resources is uniform. If this is not the case, some peers may be overloaded, since resources with the same name are mapped to the same peer. Moreover, the keys of resources having similar features are inevitably dispersed. This prevents the possibility of executing a range query efficiently, since the target resources are most likely to be located in different and remote regions of the network.

The use of swarm intelligence techniques can help to solve these issues. The basic idea is to use resource attribute(s) directly as the key, without using any hash function, and adopt an ant-inspired algorithm to sort the keys over the structure and, at the same time, distribute them to the peers in a fair fashion. This implies that resource keys are decoupled and independent from peer codes, while the two entities are strictly correlated in classical P2P systems. The idea can be applied to any kind of overlay. First, it is necessary to choose the overlay that best fits the type of attributes

[1]Available at http://self-can.icar.cnr.it.

used for indexing and searching the resources. So, a ring overlay should be chosen for resources indexed with the value of a single attribute, a multidimensional grid is preferable when resources are characterized by a set of independent attributes, a tree is indicated when resource attributes have a hierarchical structure, as in the case of XML documents.

The second step is to define, for each peer, its *centroid*. The peer centroid must represent, with a single quantity, the set of keys stored in a *local region* of the structure that comprises the peer itself and a few close-by peers. Let the distance between any two keys, say k_a and k_b, be denoted by $d(k_a, k_b)$. Let also S_k be the space of the resource keys, and $\mathcal{K}$ the set of keys stored in the considered peer and in a local region around the peer. The peer centroid $c \in S_k$ is defined as:

$$c \in S_k \mid \quad \forall c_i \in S_k, \quad \sum_{k \in \mathcal{K}} d(c_i, k) \geq \sum_{k \in \mathcal{K}} d(c, k) \tag{1}$$

Therefore, c is the value in S_k that minimizes the sum of the distances between c itself and the keys k stored in local region of the network. The value of c can be computed with a dichotomic search up to a desired precision. The peer centroids are essential to drive both the key sorting process and the resource discovery procedures. In Sections 2.1 to 2.3, devoted to specific overlays, more details will be given on the definition of *distance* and of the *local region* used for the centroid computation.

Finally, a set of mobile agents must be generated to perform the reorganization. The agents use the underlying structure to travel the system, but do not alter the structure and the way it is managed. This means, for example, that the agents can move the keys and modify the centroid values but cannot modify the peer codes neither the way peers are connected with each other, which is under the responsibility of the overlay management procedures.

The generalized algorithm performed by every agent is summarized in the pseudocode shown in Figure 1, assuming that each peer maintains a set $\mathcal{N}$ of neighboring peers, that is, peers that are connected to it in the P2P overlay structure, and a set $\mathcal{L}$ of local keys. The management of $\mathcal{N}$ is under the responsibility of the specific P2P overlay. Initially, $\mathcal{L}$ contains the keys of the resources published by the peer itself, then the keys are moved by agents through pick and drop operations, and, at a generic instant, $\mathcal{L}$ is the outcome of the process of agents moving keys.

The agent can either be *loaded*, when it is moving a key between peers, or *unloaded*. When unloaded, the agent travels the P2P overlay structure performing a random walk through neighboring peers. Whenever the agent visits a new peer, it attempts to pick a key from the peer. If a pick operation succeeds, the agent becomes loaded and starts a process aiming at dropping the key in a more appropriate peer. Thus, a loaded agent carrying a key k moves towards a region of the overlay structure that is likely populated with keys that are "similar" (or close) to k. The agent moves making jumps towards the desired region. Finally, whenever a loaded agent visits a new peer, it attempts to drop the key.

Some comments about these operations, valid for any overlay, are given in the following where, for the sake of clearness, four macro operations are identified: *Random Walk*, *Pick*, *Jump* and *Drop*.

— *Random Walk*. While the agent is unloaded (does not carry any key), it travels the network randomly, exploiting the links towards other peers that are provided by the specific overlay. The operation is repeated until a key is picked by the agent at some peer. A short time T_{mov} is waited by the agent between two consecutive steps of the random walk.

```
#code cycle executed by the agent, initially unloaded
while not (agent.loaded) do
   #start of Random Walk operation
   nextpeer = SelectNextPeer(𝒩); # select the next peer according to Random Walk rules
   wait(T_mov);
   agent.jump(nextpeer);
   #end of Random Walk operation
   c = p.centroid(); # computes centroid of p, the peer that the agent is visiting
   𝒮 = ℒ;
   while (not (agent.loaded)) and (𝒮 ≠ ∅) do
      #start of Pick operation
      k = SelectKey(𝒮); # select the key in the set 𝒮 according to some rules
      𝒮 = 𝒮 - {k};
      d = KeyDistance(k,c);
      Ppick=ComputePpick(d); # compute the prob. of picking a key that is at distance d from the centroid
      r = Random(0,1); # random number between 0 and 1, extracted with uniform pdf
      if (r < Ppick) # the Bernoulli trial has success
         agent.pick(k);
         agent.loaded=true;
      end if
      #end of Pick operation
   end while
end while # at this point the agent has picked a key and is loaded
while (agent.loaded) do
   #start of Jump operation
   c = p.centroid();
   d = KeyDistance(k,c);
   dest_peer = GetClosest(𝒩,d); # get from 𝒩 the best peer to store a key whose distance from c is d
   agent.jump(dest_peer);
   #end of Jump operation, the agent is at new peer p
   #start of Drop operation
   c = p.centroid();
   d = KeyDistance(k,c);
   Pdrop=ComputePdrop(d); # compute the prob. of dropping the carried key
   r = random(0,1);
   if (r < Pdrop)
      agent.drop(k);
      agent.loaded=false;
   end if
   #end of Drop operation
end while
# the key has been dropped, the agent is unloaded and restarts the whole cycle
```

Fig. 1. The agent algorithm. The pseudocode is executed cyclically during the life time of the agent.

— *Pick*. The goal of this operation is to pick a key that, being far from the centroid (i.e., different from the keys typically stored in the peer), does not respect the order of the keys over the structure. This operation is decided through a probabilistic process that makes use of Bernoulli experiments. Keys are considered in an order defined by a function, indicated as *SelectKey*() in the pseudocode. For each considered key k, the agent first computes the distance d of the key from the centroid c through the function $KeyDistance(k, c)$; then, it derives the pick probability *Ppick* through a proper function of distance d, and, finally, the agent performs the Bernoulli trial with probability *Ppick*. The function giving the pick probability is monotonically increasing with d, so that the larger the distance of the key from the centroid is, the higher the probability that the agent picks the key.

— *Jump*. Once the agent has picked a key, it jumps to a region of the overlay where the key is supposed to better respect the current sorting. To determine the target peer to jump to, the agent computes the distance between the carried key k and the centroid of the local peer c, again through the function $KeyDistance(k, c)$. This distance is used to estimate the correct position of the key in the overlay. Thus, the agent selects, through the function $GetClosest(\mathcal{N}, d)$, the peer *dest_peer* that, among the neighboring peers in $\mathcal{N}$, is the closest to the estimated correct position of the

key. A jump to *dest_peer* is then performed. For example, if the overlay is circular like in Chord, the distance between a key and centroid is computed as the length of the arc between the two values. If this arc is equal to 1/4 the length of the key space, the agent should reach a peer whose code is approximately distant 1/4 the circumference of the peer circle from the local peer. The agent jumps to the peer belonging to the set $\mathcal{N}$ whose code is the closest to the desired one.

— *Drop*. The objective is to drop the carried key in a peer that holds "similar" keys. As for the pick operation, a dropping is performed through a Bernoulli experiment whose success probability depends on the distance of the key from the local centroid; this time, the probability is monotonically decreasing with distance.

The described strategy is partially inspired by the basic ant algorithm introduced in Bonabeau et al. [1999]. The collective operations of agents sort both the centroids and the keys over the corresponding overlay. The sorting ensures that the basic feature of structured systems, the efficiency of discovery operations, is guaranteed. Indeed, the number of hops performed by a query message is practically the same as in the corresponding non-self-organizing P2P system, usually it is logarithmic with the number of peers. A notable improvement of the new strategy is that resource keys can be given a semantic meaning, which implies that similar resources are stored in neighbor peers and range queries can be executed efficiently. Indeed, once a search message has found a key included in the target range specified in the query, all the other keys can be discovered very rapidly, exploring the neighbor peers. This efficiency is very hard to obtain in classical P2P systems, because the keys of similar resources are spread by the hash function. The ant-inspired strategy has further interesting properties, among which: (i) it is self-organizing, as the assignment of keys is not predetermined but emerges from the combined behavior of very simple agents; (ii) it is completely decentralized, since agent operations are exclusively driven by local information; (iii) it guarantees stability: once a correct sorting has been achieved, it is quickly recovered after any perturbation – for example, peer connections and disconnections or the publication of new resources; (iv) it ensures a fair load balance, even in the presence of nonuniform key popularity.

Agents are generated and die like 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, computed on the past activity of the peer.[2] 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, $\overline{N}_p$,

$$\overline{N}_a \cong \overline{N}_p \cdot P_{gen} \tag{2}$$

It follows that the average number of agents can be regulated by modifying the value of P_{gen}. The agent load Λ can be defined as the average number of agents per second that arrive and are processed at a peer. The arrival rate of agents to peers is given by the product of the average number of agents $\overline{N}_a$ by the frequency of their movements $1/T_{mov}$, where T_{mov} is the mean of the random interval from the instant in which a peer receives an agent to the instant in which it forwards the agent to the next peer. Thus, the load on a single peer can be computed as,

$$\Lambda = \frac{\overline{N}_a}{\overline{N}_p \cdot T_{mov}} \approx \frac{P_{gen}}{T_{mov}} \tag{3}$$

[2]If the peer enters the network for the first time, it uses the average connection time of a neighbor peer.

From the evaluation of Self-Chord [Forestiero et al. 2010] and Self-CAN [Giordanelli et al. 2011], it was found that the sorting process converges quickly enough by setting T_{mov} to 5 seconds and P_{gen} to 1.0. In such a scenario each peer receives and processes about one agent every 5 seconds, which is an acceptable load, since pick and drop operations are very simple. Moreover, it should be noticed that Λ does not depend on the frequency of peer joinings and disconnections nor on the network size, therefore it is stable even in the presence of abrupt environmental changes. For example, if several resources are published in a short interval of time by a new peer, the resource keys will be picked by the agents that pass by the new peer. Conversely, in a classical structured system the new keys must be immediately placed in the corresponding peers, which may cause a high load during the time interval necessary to complete this process.

In the following sections, we briefly describe how the ant-based strategy can be adapted to three specific overlay: a ring, a multidimensional grid, and a tree.

2.1. Ring Overlay

If a resource can be characterized by the value of a single attribute, say an integer between 0 and V-1, the most convenient structure is a ring, like that used in Chord, in which peers connections are established both between adjacent peers and through the shortcuts of the finger tables maintained by peers. The distance between two keys is defined as the length of the shorter of the two arcs that separate the keys in the circular space. For example, with $V = 10$, $d(3, 7) = 4$ and $d(9, 1) = 2$, because the key value 0 is the successor of 9. The local region centered at a peer includes the peer itself, a number of peers on the left, and the same number of peers on the right. The centroid is computed using expression (1).

For this scenario, agent operations can be instantiated as follows.

— *Random Walk*. The unloaded agent travels from adjacent to adjacent peer. The direction, clockwise or anticlockwise, is chosen randomly.
— *Jump*. The position of the target peer, that is, its code, is obtained with a simple proportion between the distance between the key and the local centroid and the distance between the local peer and the target peer. The agent examines the peers of the finger table and jumps to the one that is the closest to the target peer.
— *Pick* and *Drop*. Pick and drop probability functions are, respectively, directly and inversely proportional to the distance between the considered key and the local centroid. More details are given in Forestiero et al. [2010].

Once the keys have been sorted, the resulting overlay appears as depicted in Figure 2, in a very simplified scenario in which the key values are between 0 and 7, and the overlay contains 16 peers. For each peer, the figure shows some of the keys stored locally and the centroid value. Notice that keys are sorted, and that their values have no relationship with the peer codes, which are not reported. As a consequence, also the centroids are sorted. This ensures that discovery procedures can be executed in log time, since the shortcuts of the peer finger tables allow the search space to be halved at each jump of the search message. This aspect is extensively discussed in Forestiero et al. [2010].

2.2. Multidimensional Overlay

A multidimensional overlay is preferable when a resource is conveniently characterized by several independent attributes. Examples are an OLAP environment [Pedersen and Jensen 2001], in which resources are characterized by attributes like time, space, price, etc., or a Grid/Cloud infrastructure in which the hosts are characterized by CPU, memory, and bandwidth.

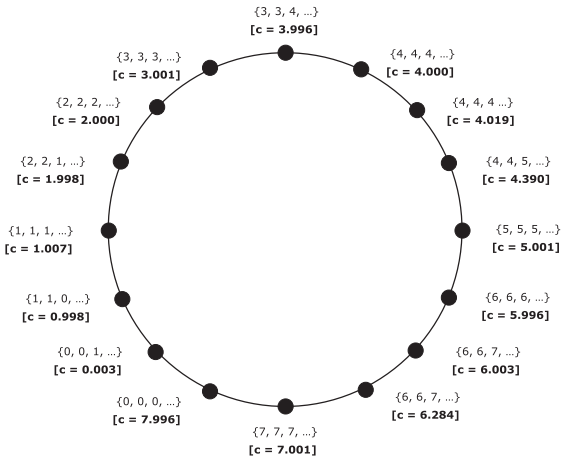

Fig. 2. Centroid ordering in a ring overlay.

Since the following sections of this article are devoted to this specific case, many details about agent operations and probability functions are given later. The distance between two keys is computed as a form of "Manhattan" distance and will be defined in Section 3.2. The local region for the centroid computation includes the local peer and the peers that are adjacent along the different dimensions, in both directions. As an illustrative example, let us consider the case that the key space is bidimensional and keys assume integer values in the range [0, 15] for each dimension, with a circular ordering defined along each dimension. For an example of centroid computation, let us assume that the keys stored in the local region are the following: (1, 15), (3, 14), (5, 0) and (3, 1). In this case, the centroid of the central peer 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. Agent operations are specialized as follows.

— *Random Walk*. The agent travels from adjacent to adjacent peer, randomly choosing both the dimension and the direction.
— *Pick*. As usual, a key should be picked when its distance from the local centroid is high (in the example, with the centroid equal to (3,15.5), a key (11, 7) should be picked with very high probability, whereas a key (3, 15) should be left on the peer).
— *Jump*. The agent moves along the dimension that allows to minimize the distance between the carried key and the centroid of the target peer. 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 (in the example, key (0, 15) should be moved towards a "predecessor" peer along the first dimension, whereas a key (3, 4) should be moved towards a "successor" peer along the second dimension).
— *Drop*. An agent that arrives at a new peer should drop the carried key if its value is similar to the peer centroid, and vice-versa (in the example, an agent that arrives at the local peer carrying the key (3, 15), previously picked on another peer, should drop the key).

Figure 3 shows the values of peer centroids obtained at the end of an experiment executed with the Self-CAN prototype, in the case of a 2-dimensional overlay with 16 peers. It can be noticed that centroid values are sorted along both dimensions, which

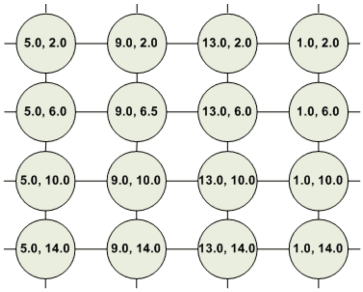

Fig. 3. Centroid ordering in a 2-dimensional overlay with 16 peers.

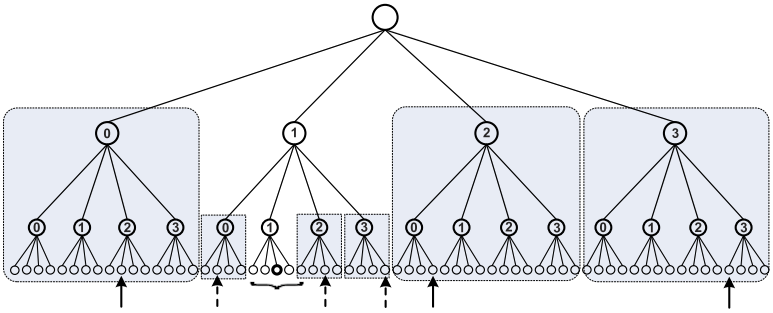

Fig. 4. Pointers of the routing and leaf tables in a Pastry overlay.

means that the values of single keys (not reported) are also sorted. Yet, there is no predetermined association between key values and peers. A discovery message, issued to find a target key, is easily driven following the gradient of centroid values, towards the peer whose centroid is equal or as close as possible to the target key. As Section 4 will discuss, the target key is most likely located in that peer.

2.3. Tree-Based Overlay

A tree-based overlay should be chosen when the resources are conveniently characterized by hierarchical indexes. This may be the case of distributed XML databases: an XML document, or a section of a document, can be indexed and searched through a hierarchical index built following the path from the root to the document or section [Harder et al. 2007].

A convenient overlay for such a case is provided by Pastry [Rowstron and Druschel 2001] or a similar tree-based P2P system. In Pastry, a peer is characterized by a code of B digits, each of which may assume one out of b possible values. The peers are organized making reference to a tree structure ordered according to the peer code. In the example shown in Figure 4, the number of tree levels is $B = 3$. A vertex with value i at level l has the lth digit equal to i; the vertex has $b = 4$ children. Peer codes corresponds to leaves of the tree; thus, two peers whose closest ancestor is at level i share the first i digits of the code. The peer *routing table* contains pointers to peers whose codes share a common prefix, of every length, with its code. With reference to the example shown in the figure, the peer with the marked border has code 112. For this peer, the first row of the routing table contains pointers to at least one peer (a leaf of the tree) for each of the big shaded rectangles; in other words, the row contains pointers to three peers whose codes begin with digits 0, 2, and 3. Three peers of this kind–specifically, peers 021, 203, and 331–are indicated by the continuous-lined arrows. Analogously, the

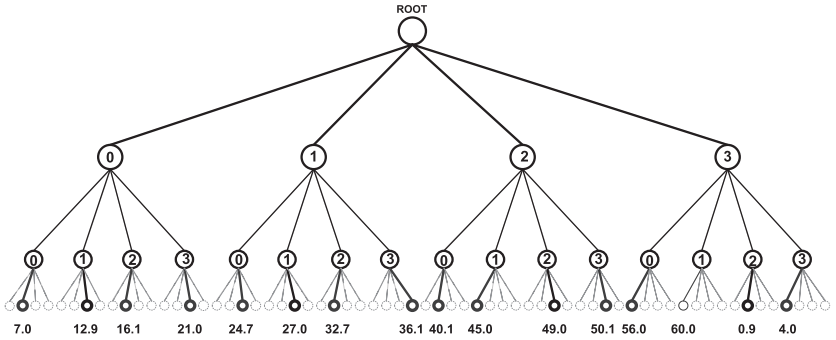

Fig. 5. Centroid ordering in a tree-based overlay.

second row of the routing table contains pointers to peers included in the small shaded rectangles, whose codes start with digits 10, 12, and 13. Example of these peers–101, 122, and 133–are indicated by the dashed-lined arrows. In addition, the *leaf table* contains direct pointers to some close-by peers, for example, those embraced by the curly bracket in the figure.

Also in this kind of overlay, the ant-inspired algorithm can be used to distribute and sort the XML document keys over the structure. The advantage is that key values can be the digital representations of document positions in the XML hierarchy. This may speed up the execution of range queries (e.g., all the papers published in a given journal within a time interval), because the target resources are most likely located in neighbor peers.

The distance between two keys may be defined as the difference between the key values. The local region for the computation of a centroid may be defined as the set of peers that share the same ancestor in the tree representation. In the Random Walk phase, the agent randomly follows one of the pointers of the leaf table. To execute the Jump operation, the peer carrying a key computes the distance from the key and the local centroid, projects it to a corresponding distance in the space of peer codes, and determines the target peer. Then, the agent exploits the leaf or the routing table to get to the peer whose code is as close as possible to that of the target peer. Pick and Drop operations are executed in the usual way. Figure 5 shows the ordering of centroids at the end of an experiment executed on a Pastry overlay with 16 connected peers.

3. THE SELF-CAN P2P SYSTEM

After introducing the generalized ant-inspired approach, we focus on its application to a multidimensional overlay, specifically to that defined by the CAN system [Ratnasamy et al. 2001]. The resulting system is called Self-CAN. The basic guidelines have already been introduced in Section 2.2 using an example of a 2-dimensional overlay. In this section, we first recall some fundamental properties of the CAN system, and discuss the mapping between resource attributes and overlay dimensions in Self-CAN. Then, we define the Pick and Drop probability functions that drive the corresponding agent operations, and give some interesting analytical insights into the behavior of successive Pick attempts. Finally, we describe the Self-CAN discovery procedure, with specific reference to the execution of range queries.

3.1. Basic Properties of CAN and Dimension Tuning in Self-CAN

In CAN, each peer that connects to the network is assigned an index whose value is a point in a multidimensional space, randomly computed with a hash function. The

process starts by assigning the whole multidimensional space to a single peer, then the network evolves and the space is 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 multidimensional space, using another hash function, and the key is delivered 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 multidimensional 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 Ratnasamy et al. [2001] for more details).

— The multidimensional space is toroidal to avoid border effects; without loss of generality, the values assigned to peer indexes are in the range (0, 1) for each dimension.
— Each peer is connected to $2D$ neighbor peers, that is, 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 $\log_2 N_p$.

In Self-CAN, there is no need to use a hash function to compute the key of a resource: each resource is assigned a multidimensional key, letting the key represent the main resource characteristics. The most convenient choice would be to associate each significant attribute of the resource with a component of the key, and with a dimension of the multidimensional space: in this case, there would be a perfect matching between the Self-CAN overlay and the resource key components. The drawback may be that the number of dimensions D, when set in this way, may be too small to guarantee logarithmic search time. In this case, some simple solutions are possible to reduce search complexity. The first one is to map a single attribute to a few dimensions. For example, an attribute with 256 possible values may be mapped to two subattributes having 16 values each, by coding the value of the original attribute with two sets of 4 bits. This means that a monodimensional range query may need to be converted in a two-dimensional one. Fortunately, though, multidimensional range queries are managed efficiently in Self-CAN, as discussed in Section 4.4. A second solution is to add a few long links between peers along a specific dimension. In Kleinberg [2000], it is shown that the addition of a long link, in the case that its length is generated with a harmonic probability distribution, ensures that the average path length becomes $O(log^2 n)$, where n is the number of nodes in a circle. This result is generalized in Symphony [Manku et al. 2003]: with k links, the path length is $1/k \cdot O(log^2 n)$. It should be noticed that, in practical scenarios, the addition of a few long links is much more efficient in Self-CAN than in other systems like Chord, Pastry, and Mercury. In these systems, long links are used to explore the whole network, while in Self-CAN they are only used to traverse the peers that are aligned along a single dimension. For example, if a Self-CAN network with 125,000 peers is structured in three dimensions, long links can be used to drive discovery requests through only 50 peers per dimension on average.

In the rest of the article, in order to simplify the discussion, we assume that the number of dimensions is such that the logarithmic search time is guaranteed, or that one of the strategies previously discussed is applied to achieve the same result.

3.2. Pick and Drop Probability Functions in Self-CAN

As discussed in Section 2, Pick and Drop operations are executed by agents on the basis of Bernoulli trials whose probabilities depend on the distance between the key k under evaluation and the centroid c of the local peer. This distance, $d(k, c)$, is defined as:

$$d(k, c) = \frac{1}{D} \sum_{i=1}^{D} \frac{\Delta_i}{L_i/2}, \tag{4}$$

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, that is, 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 (it can be seen as the length of the semi-circle in the space of keys along that dimension) and it is used in the fraction denominator to normalize the distances over the different dimensions. The value of $d(k, c)$ is actually the normalized "Manhattan" distance between k and c. The *similarity* between k and c, $f(k, c)$, is defined as $1 - d(k, c)$, and ranges between 0 (minimum similarity) and 1 (k and c have exactly the same value).

When an unloaded agent arrives at a new peer following its Random Walk, it evaluates the *pick probability function*, which for a local key k is defined as:

$$P_{pick}(k) = \frac{\alpha_p}{\alpha_p + f(k, c)} \quad \text{with } 0 \leq \alpha_p \leq 1. \tag{5}$$

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, unless otherwise stated, as in the base ant algorithm introduced in Bonabeau et al. [1999].

While carrying a key,[3] the agent performs the Jump operation. It 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, the agent tries to drop the key with a Bernoulli trial with probability defined by the *drop probability function*:

$$P_{drop}(k) = \frac{f(k, c)}{\alpha_d + f(k, c)} \quad \text{with } 0 \leq \alpha_d \leq 1, \tag{6}$$

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

[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.

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.

3.3. Analysis of Pick Attempts

During the Pick phase, the agent associates a pick probability with each key held by the local peer. Pick attempts are performed one at a time and the agent leaves the peer as soon as a key is picked. The order in which such attempts are made is significant. This aspect is examined here.

Denote by $A(k)$ the *actual* pick probability of k, that is, the actual probability that the agent leaving the peer has picked and, thus, carries key k. Assume that the peer holds one key per class, with distance from the centroid equally spaced from 0 to D_M at intervals of length d. This means that there is no key with distance larger than D_M from the centroid: the farthest away key has distance D_M, the farthest but one key has distance $D_M - d$, the following one has distance $D_M - 2d$, and so on. The agent considers the keys in their reverse order with respect to their distance from the centroid, starting from the most distant key, and uses the pick probability defined in (5) for each attempt. We also number the keys in reverse order, k_0 being the farthest one, k_i being the one with distance $D_M - id$.

The first key, k_0, is actually picked with probability $P_{pick}(k_0)$, computed as in (5) with the distance equal to the maximum distance D_M.

$$A(k_0) = \frac{\alpha_p}{\alpha_p + f(k_0, c)}.$$

The following key, at distance equal to $D_M - d$, is actually picked with probability

$$A(k_1) = P_{pick}(k_1)\left(1 - P_{pick}(k_0)\right)$$

and key k_i is actually picked with probability

$$A(k_i) = P_{pick}(k_i)\prod_{j=0}^{i-1}\left(1 - P_{pick}(k_j)\right).$$

The effectiveness of the agent action is much higher if keys are ordered. Figure 6 quantifies this comparison. Consider four cases with D_M varying between 0.2 and 0.8; in each case, $N_c = 11$ keys are equally spaced in $[0, D_M]$; $\alpha_p = 0.1$. Besides the curves of the actual pick probability obtained by ordering keys according to their distance from the centroid, the figure shows the curves obtained from a random ordering. In the latter case, the agent randomly chooses a key and decides whether to pick it; if the key is not picked, the agent attempts with another key, randomly chosen among the remaining keys. The figure shows that ordering keys remarkably differentiates the

[4]The values of α_p and α_d only affect the speed 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.

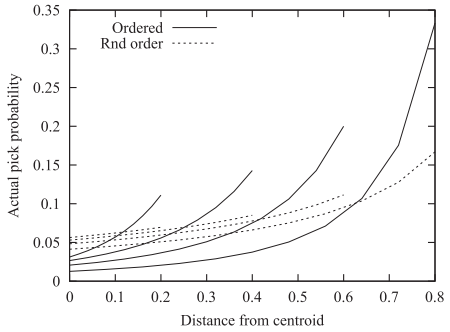

Fig. 6. Actual pick probability versus distance between the key and the centroid for different values of D_M, with random or ordered pick attempts; $\alpha_p = 0.1$.

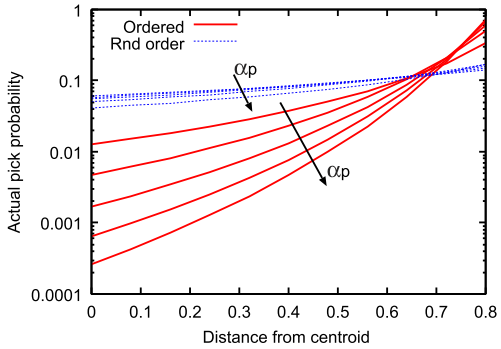

Fig. 7. Actual pick probability versus distance between the key and the centroid for various values of α_p, with random or ordered pick attempts, $D_M = 0.8$.

actual pick probability of keys that are at different distances from the centroid, the probability of choosing farther keys being much higher than the probability of choosing keys that are close to the centroid. This differentiation ensures that the speed with which the peer gets rid of the keys that are far from the centroid is very high.

Figure 7 shows, instead, the impact of the value of α_p, which varies between 0.1 and 0.5, for $D_M = 0.8$. The impact of α_p is quite limited in the random order case. With ordered pick attempts, the differentiation among the values of the actual pick probability increases with the value of α_p. It was observed that acting on the ordering of pick attempts is far more effective than tuning α_p. The setting of α_p does not have much impact on the key reordering process, it just marginally acts on the process velocity.

3.4. Discovery Procedures in Self-CAN

Discovery procedures are defined to serve *punctual* and *range* queries. The purpose of a punctual query is to find the resources belonging to a specific *class*, that is, which have been associated with a specific value of the multidimensional key. This is a typical problem in many distributed environments (e.g., Grids and large Cloud or multi-Cloud frameworks), 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.

The discovery algorithm for the case of punctual queries is very simple. At every step, the peer that receives the search message (or, at the first step, the peer that

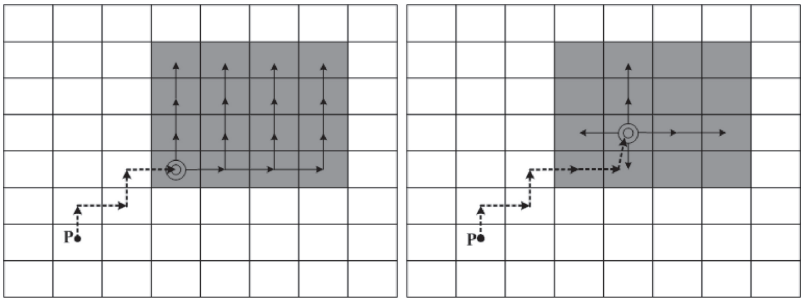

Fig. 8. Path of range queries with the two defined strategies: sweep up (left), explosion (right).

generates the request) computes the normalized distances, along the different dimensions, between 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.

Self-CAN can also efficiently manage range queries, thanks to the sorting of keys over the multidimensional 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 left 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 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.[5] The query path is depicted in right part of Figure 8. Of

[5]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.

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.

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.

In the experiments, the P_{gen} probability is set to 1.0, which means that each new or reconnecting peer issues one mobile agent. 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. 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 is used as a time unit and the performance results versus time are reported accordingly.

Experiments were performed to evaluate different aspects: the capacity of the algorithm to fairly distribute and sort the resource keys over the multidimensional space, the efficiency and effectiveness of resource discovery operations, for both punctual and range queries, the traffic load, the dynamic behavior. All these aspects are described in the rest of this section, starting with analyzing the balance of load among the peers and the robustness of Self-CAN to the distribution of key popularity.

4.1. Load Balancing and Impact of Key Popularity

An important feature 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 multidimensional 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. Self-CAN adaptively distributes the keys over the peers irrespective of the distribution of key popularity, which is often nonuniform [Goh et al. 2005]. Furthermore, Self-CAN is also robust to the fact that key popularity changes over time, since in P2P systems new objects often become the most popular quite rapidly [Gummadi et al. 2003].

To illustrate the behavior of Self-CAN with nonuniform popularity, we set two simple experiments, for a network with $D = 2$ and $N_p = 64$. In the first experiment, the keys can have values from 0 to 7 for each dimension, the centroids are assumed to be uniformly spaced, 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 9. 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

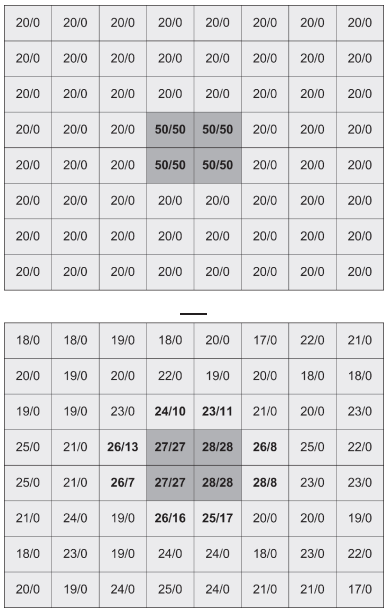

Fig. 9. Distribution of keys for the load balancing test, at the beginning of the process (top) and in steady situation (bottom).

(0, 0), and stores 20 keys with the same value. The fourth peer of the fourth 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, and depicted in the bottom part of Figure 9, 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 peers, 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 one 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 probability functions.

To show this, we varied the values of either α_p or α_d, setting the other parameter to the value $\alpha_p = 0.2$ or $\alpha_d = 0.5$. For each test, we computed the ratio L_h/L_m, where L_h is the average load (number of keys) of the four central peers, and L_m is the average load of all the peers. Figure 10 shows the values of the ratio in steady conditions, which are much lower than at the beginning (when the ratio is about 2.5). Notice that the trend is not monotonic and the values of α_p and α_d can be tuned to optimize the load balancing.

For the second experiment, instead of using the ad-hoc distribution adopted for the first experiment, we used a more realistic distribution of key popularity. A statistical study on the resources shared by a P2P system is presented in Goh et al. [2005] and is founded on real data provided by Gnutella and Napster. The study shows that the popularity of songs, within a genre, follows a Zipf distribution. Indeed, this assumption is shared by many works on P2P and Web systems. Accordingly, for the same scenario as shown previously, with $D = 2$ and $N_p = 64$, we assumed that each peer publishes 50

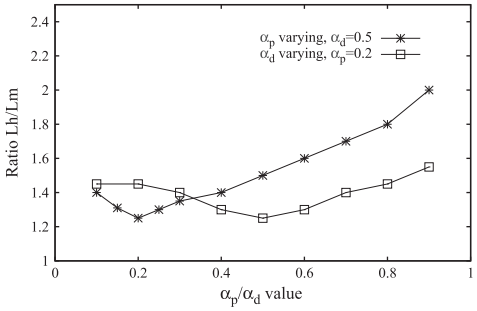

Fig. 10. Trend of the ratio L_h/L_m versus α_p and α_d.

(0,0) 25/60	(0,0) 13/38	(1,0) 9/54	(2,13) 5/35	(3,13) 6/41	(6,0) 6/38	(9,0) 10/53	(14,0) 10/51
(0,0) 16/60	(0,0) 12/51	(2,0) 20/61	(3,0) 24/59	(4,0) 13/55	(7,0) 18/70	(8,0) 9/56	(11,1) 9/60
(0,2) 20/69	(1,2) 12/40	(2,1) 11/43	(3,1) 10/43	(5,1) 10/44	(6,1) 10/41	(9,2) 12/60	(11,3) 5/26
(0,6) 12/54	(1,4) 13/57	(2,2) 13/57	(3,3) 10/38	(5,2) 11/45	(7,2) 10/64	(10,4) 8/46	(12,5) 9/54
(0,6) 11/49	(0,5) 7/43	(2,5) 15/64	(3,4) 8/44	(4,4) 9/43	(7,4) 7/63	(10,5) 9/53	(12,6) 5/35
(0,8) 9/49	(1,9) 8/56	(2,7) 7/47	(4,5) 8/42	(6,6) 7/53	(7,7) 6/48	(11,7) 7/70	(12,8) 6/54
(0,10) 7/44	(1,11) 8/42	(1,9) 4/36	(3,8) 8/47	(4,7) 7/50	(6,9) 6/47	(9,10) 7/70	(12,12) 6/54
(0,0) 14/37	(0,0) 9/45	(0,12) 5/43	(3,10) 5/38	(4,11) 9/65	(6,10) 13/61	(9,13) 5/52	(12,13) 4/34

Fig. 11. Distribution of keys for the test with Zipf popularity distribution. For each peer, the snapshot shows the coordinates of the most popular key and, below, the number of such keys and the total number of keys stored locally.

keys that can have values from 0 to 15 for each dimension, and that the frequency of value i, for each dimension, follows the Zipf distribution: it is equal to $c/(i+1)^{(1-\theta)}$. In this expression, c is a normalization constant to ensure that the sum of frequencies is equal to 1, and the parameter θ, which in general can assume values between 0 and 1, is here set to 0.5. Thus, the most popular value of each dimension is 0, and popularity is monotonically decreasing so that 15 is the least popular value. This means, for example, that about 2.2% of the keys have value (0, 0), while only 0.14% of the keys have values (15, 15). A snapshot of the network, taken after the keys have been sorted and redistributed, is shown in Figure 11. For each peer, we report the value of the most popular key held in the peer, in the form (k_x, k_y), the number of such keys N_{max} and the total number of keys stored locally, N_{tot}: the last two quantities are separated by the "/" character. Beyond confirming that the keys have been actually sorted,[6] the figure shows that the load is balanced, thanks to the fact that the most popular keys are spread in a few adjacent peers. For example, the key (0, 0), that is the key with the highest popularity, is also the most popular key in 6 peers (4 in the top left corner and 2 in the bottom left corner), and the value of N_{max} is higher than the average in such

[6]Clearly, the centroids are sorted as well. The centroid of each peer either coincides or is very similar to the most popular key stored in the peer.

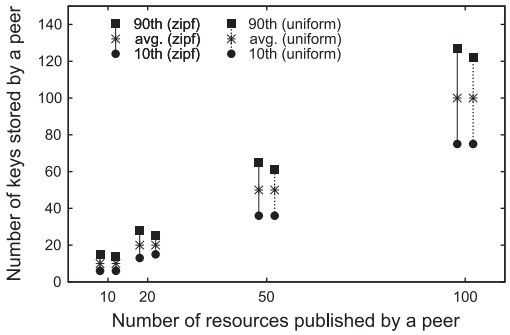

Fig. 12. Average, 10th and 90th percentile of the number of keys stored by peers, when using the Zipf and the uniform popularity distribution, with different values of the number of resources published by a peer.

peers. In contrast, in none of the peers, the most popular key has value 15 in any of the two dimensions.

To analyze the possible impact of the popularity distribution, the same experiment was executed assuming a uniform popularity of keys. Then, the average, the 10th and the 90th percentile of the number of keys stored by a peer were reported for the two experiments. The comparison, reported in Figure 12, shows that the load distribution is very similar: the 90th percentile is only moderately higher when using the Zipf distribution.

It is also interesting to consider the different behaviors of Self-CAN and CAN in the case of Zipf popularity. In CAN, the peers tend to equally share the space of keys, which means–for the scenario previously described–that a peer may be required to manage the keys with values (0, 0), (0, 1), (1, 0) and (1, 1), and another peer the keys (14, 14), (14, 15), (15, 14) and (15, 15). Considering the Zipf popularity of keys, the former peer would manage about 210 keys, while the latter less than 20 keys: clearly, the resulting load balance would be much worse than the one shown in Figure 11.

4.2. Analysis of the Reordering Process

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

To valuate the first characteristics effectively, we focus now on uniformly distributed popularity of the keys and assess whether the distance between peer centroids is also uniform. We compute the average distance, in the space of resource keys, between two "consecutive" centroids, that is, the centroids of two adjacent peers. As definition of distance, we consider the Manhattan distance. 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 distance between the centroids of two peers that are adjacent along dimension i must be comparable to L_i/N_i.[7]

Let us consider the case in which the space of keys is a hyper-cube, that is, 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, being each class associated to a specific value of the multidimensional key. In this case, the expected Manhattan

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

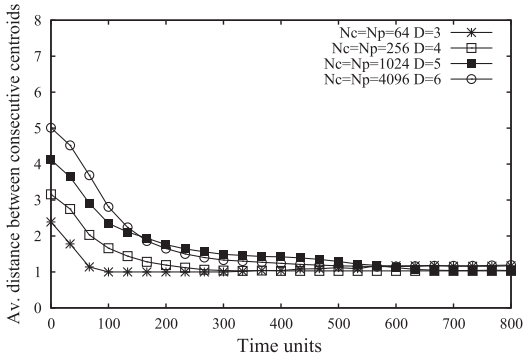

Fig. 13. Average Manhattan distance between consecutive centroids in networks with different size.

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 13 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 $(\log_2 N_p)/2$. In these experiments, 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. Each coordinate of the key is generated randomly in the range between 0 and $\sqrt[D]{N_c} - 1$. At the beginning, key values are distributed randomly in the network. At time 0, the sorting algorithm is started and the keys are sorted through the operations of Self-CAN agents.

The figure shows that the average value of the centroid distance rapidly converges to the expected value, confirming that the centroids are reordered correctly and efficiently. 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 is 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:

$$H_p = 1 - \frac{\sum_{(k_x,k_y)} d(k_x, k_y)}{n_k} \tag{7}$$

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 14, that shows the overall homogeneity function computed during the experiments described previously. 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

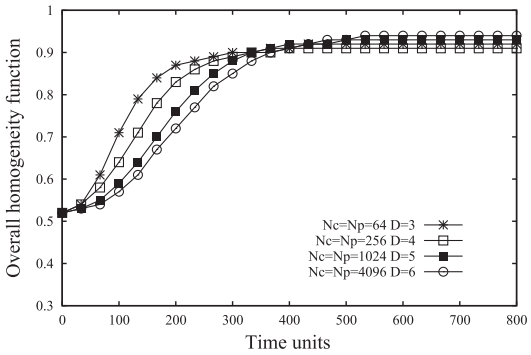

Fig. 14. Overall homogeneity function in networks with different sizes.

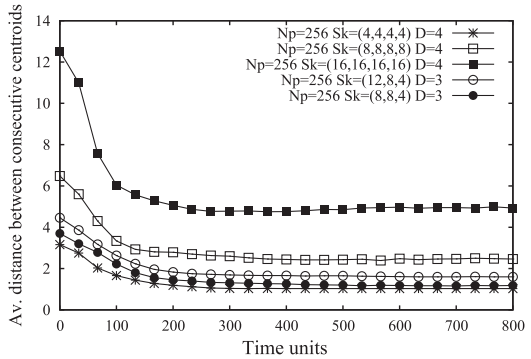

Fig. 15. Average Manhattan distance between consecutive centroids in a network with fixed size and different settings for the key space vector S_k.

is hardly affected by the network size, which is a sign of the good scalability properties of Self-CAN.

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 15 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.

4.3. Performance of the Discovery Procedure

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. If this is true, a search for a target key can be converted in a search for a peer centroid. Figure 16 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

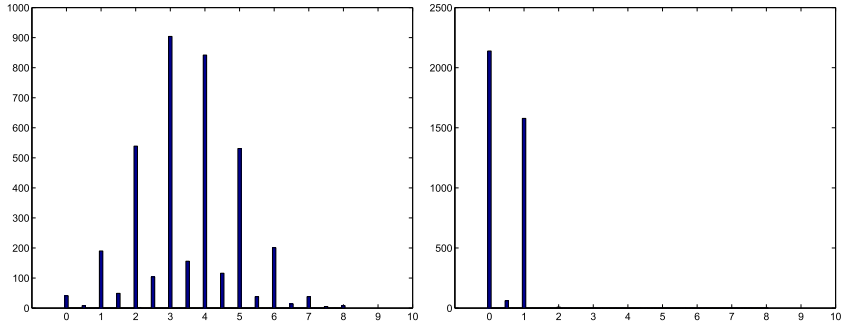

Fig. 16. Histogram of the Manhattan distance between a key and the local centroid: before the start of the sorting process (left plot), and in a steady condition (right plot).

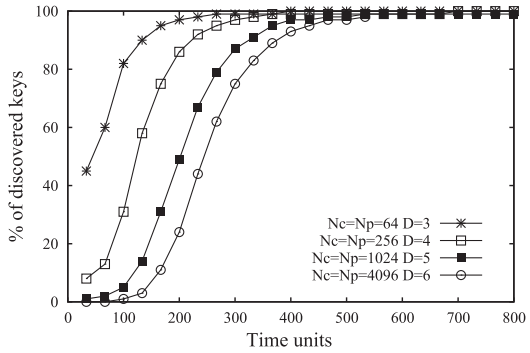

Fig. 17. Percentage of discovered keys in networks with different sizes.

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 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, that is, one forth of the facet length measured in the key space. In fact, the distribution of the distances between key and centroid (left plot of Figure 16) is centered on a value slightly lower than 4, and has a typical Gaussian shape. In a steady condition (right 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.

Figure 17 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. In these experiments, the key space is a hypercube, and the number of admissible values of keys is 4 for each

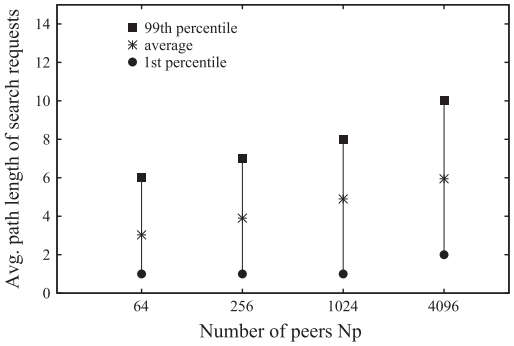

Fig. 18. Average, 1st and 99th percentile of the path length of discovery procedures in networks with different size.

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 18, 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. 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. These experiments were repeated assuming a Zipf distribution of key popularity for each dimension. The statistics on path length are practically identical, so they are not reported.

4.4. Performance of Range Queries

In Section 3.4, two approaches were defined to serve range queries: *sweep up* and *explosion*. 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 ith element specifies the size of the range of values that are searched for in dimension i. Two cases are considered. In the first one, $R_k = (3, 3, 1, 1, 1)$ 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 19 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 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. The appropriate approach should be chosen depending on the application requirements: for example, in an OLAP analysis [Pedersen and Jensen 2001] 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 or Cloud hosts with given requirements.

As a conclusive remark, both strategies are feasible because the ordered keys are allowed to have semantic values, associated to resource attributes. In classical structured P2P systems, the key values are spread by hash functions, therefore range

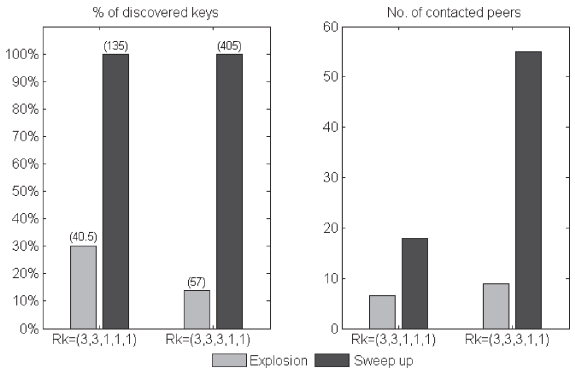

Fig. 19. Percentage of discovered keys (the absolute number is also indicated on top of the bars) and number of contacted hosts with sweep-up and explosion approach, for two types of range queries. The network has 1024 peers and the key pattern is S_k = (4,4,4,4,4).

queries are usually 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.

4.5. Network 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 network load if many resources are published in a short interval of time. In Self-CAN, the network load is invariant because a new peer does not need to perform any additional operation[8]: the keys of the new resources will be picked by the agents that pass by this peer, what changes is the frequency of successful pick and drop operations performed by the agents. Moreover, the agent load experienced by a single peer does not depend on the network size (see the evaluation of load for the generalized ant-inspired approach in Section 2), which confirms the scalability properties of Self-CAN.

The results discussed in Sections 4.2 and 4.3 show 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 4.2 and 4.3, a new peer joined the network when ordering of keys can already 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.

A set of specific experiments was performed to evaluate the dynamic behavior of Self-CAN in the case of a very intense node churn: once the reordering process has reached a steady condition, a perturbation is generated by simulating the simultaneous arrival of a large number of new peers, each with 15 new resources on average. In these tests, the number of dimensions is 5, the key space is S_k = (4, 4, 4, 4, 4), and the initial number of peers N_p is 1000. After 830 time units, a number of new peers, specified as a percentage P_{join} of N_p, join the network. The value of P_{join} was set to 5%,

[8]Besides the operations related to the overlay management, such as the update of the list of neighbors.

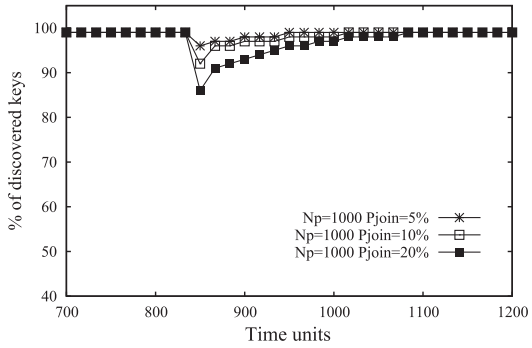

Fig. 20. Percentage of discovered keys after a node churn. In a network with 1000 peers, a percentage P_{join} of new peers join the network simultaneously.

10%, and 20%, corresponding to the simultaneous arrival of 50, 100, and 200 peers. Figure 20 reports the percentage of target keys discovered by punctual queries before and after the node churn. The figure shows that the system returns to the stable condition very rapidly, just through the ordinary work of agents, that is, without any increase either in the network traffic or in the processing load of peers.

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. Clearly, a mechanism of this kind is needed in any P2P system, it is not a peculiar requirement of Self-CAN.

Finally, Self-CAN is also 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 [Iamnitchi and Foster 2004; Taylor 2004] 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).

Self-organization algorithms are often inspired by the behavior of biological systems, such as insect swarms and ant colonies [Bonabeau et al. 1999]. The basic feature of these algorithms is that they allow complex forms of swarm intelligence to emerge, at a high level, from the combination of simple operations performed by a multitude of agents at the low level. These algorithms are already exploited in a wide variety of domains, ranging from robotics to power and telecommunication systems.

Recently, swarm and bio-inspired algorithms proved capable of considerably improving the performance of P2P computer systems [Guéret et al. 2007; Ko et al. 2008]. Two interesting examples are Anthill and BlatAnt. The Anthill project [Babaoglu et al. 2002] is tailored to the design, implementation and evaluation of P2P applications based on multiagent 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 [Brocco et al. 2010]. 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 [Forestiero et al. 2010], ant algorithms proved capable of triggering a self-organization behavior also in ring-structured P2P overlays.

As opposed to the mentioned systems, Self-CAN efficiently supports complex and range queries on multiple attributes. This is indeed a very tough issue in P2P systems [Cheema et al. 2005], 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 [Albrecht et al. 2008; Datta et al. 2005], or increasing the traffic load by issuing a number of subqueries [Andrzejak and Xu 2002]. The Squid discovery protocol [Schmidt and Parashar 2004] uses a dimension-reducing technique, the space-filling curve (SFC), to map multiattribute keywords to a monodimensional space, that is, a ring overly 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 multiattribute 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 [Bharambe et al. 2004], 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.

In the introductory section, it was anticipated that our article follows the research avenue, discussed in Jelasity et al. [2009], which tries to develop flexible and configurable protocols that are capable of addressing any kind of overlay network. The same paper is related to our work in another way: it presents an algorithm, namely `T-Man`, which can be used to create a large class of overlay networks from scratch, including rings, trees and toruses. Therefore, `T-Man` can be to used to create the type of overlay that better adapts to the specific application domain for which our ant-inspired approach is tailored: a ring for organizing single-attribute resources, a multidimensional torus for multiattribute resources, etc.

The HyperCBR [Castelli et al. 2008] system adopts the content-based routing approach: routing is based on message content rather than destination address. HyberCBR relies on a multidimensional space where subscriptions and published events are routed along distinct partitions, for example, along different dimensions of the overlay. A resource discovery procedure succeeds when subscriptions and events intersect in

at least one node. The main merit of this strategy is that it can exploit the power of pattern-based search, for example, regular expressions.

6. CONCLUSION

This article has presented a generalized ant-inspired approach that can be used to sort the resource keys over any kind of P2P overlay. The process sorting, performed through the pick and drop operations of mobile agents, is statistically driven, self-organizing, and decentralized. The distribution of keys over the peers is not constrained by the values of peer codes, as in ordinary P2P systems. This enables the possibility of preserving the values of significant resource attributes into the resource keys, instead of generating the latter with hash functions. The advantages of this approach are numerous, ranging from a more efficient execution of complex queries to improved behavior in terms of adaptivity and load balancing. This article has described how the approach can be applied to several types of overlays, and has focused on the specific case of Self-CAN, a self-organizing P2P system based on a multidimensional structure. Thanks to its properties, specifically to its capacity of serving multidimensional range queries, Self-CAN is particularly well suited to large Grid and Cloud environments.

REFERENCES

ALBRECHT, J., OPPENHEIMER, D., VAHDAT, A., AND PATTERSON, D. A. 2008. Design and implementation trade-offs for wide-area resource discovery. *ACM Trans. Internet Tech. 8*, 4, 1–44.

ANDROUTSELLIS-THEOTOKIS, S. AND SPINELLIS, D. 2004. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv. 36*, 4, 335–371.

ANDRZEJAK, A. AND XU, Z. 2002. Scalable, efficient range queries for grid information services. In *Proceedings of the 2nd IEEE International Conference on Peer-to-Peer Computing (P2P'02)*. IEEE Computer Society, 33–40.

ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. 2010. A view of cloud computing. *Comm. ACM 53*, 4, 50–58.

BABAOGLU, O., MELING, H., AND MONTRESOR, A. 2002. Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*. IEEE Computer Society, 15–22.

BHARAMBE, A. R., AGRAWAL, M., AND SESHAN,. S. 2004. Mercury: Supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev. 34*, 4, 353–366.

BONABEAU, E., DORIGO, M., AND THERAULAZ, G. 1999. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press.

BROCCO, A., MALATRAS, A., AND HIRSBRUNNER, B. 2010. Enabling efficient information discovery in a self-structured grid. *Future Gen. Comput. Syst. 26*, 838–846.

CASTELLI, S., COSTA, P., AND PICCO, G. P. 2008. HyperCBR: Large-scale content-based routing in a multidimensional space. In *Proceedings of the 27th IEEE International Conference on Computer Communications (INFOCOM'08)*. 1714–1722.

CHEEMA, A. S., MUHAMMAD, M., AND GUPTA, I. 2005. Peer-to-peer discovery of computational resources for grid applications. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*. 179–185.

DATTA, A., HAUSWIRTH, M., JOHN, R., SCHMIDT, R., AND ABERER, K. 2005. Range queries in trie-structured overlays. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing (P2P'05)*. IEEE Computer Society, 57–66.

DE CANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. 2007. Dynamo: Amazon highly available key-value store. Tech. rep. http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf, Amazon.

FORESTIERO, A. AND MASTROIANNI, C. 2009. A swarm algorithm for a self-structured P2P information system. *IEEE Trans Evol. Computat. 13*, 4, 681–694.

FORESTIERO, A., LEONARDI, E., MASTROIANNI, C., AND MEO, M. 2010. Self-chord: A bio-inspired P2P framework for self-organizing distributed systems. *IEEE/ACM Trans. Netw. 18*, 5, 1651–1664.

FOSTER, I. AND KESSELMAN, C. 2003. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, CA.

GIORDANELLI, R., MASTROIANNI, C., AND MEO, M. 2011. A self-organizing P2P system with multi-dimensional structure. In *Proceedings of the 8th IEEE/ACM International Conference on Autonomic Computing (ICAC'11)*.

GOH, S. T., KALNIS, P., BAKIRAS, S., AND TAN, K.-L. 2005. Real datasets for file-sharing peer-to-peer systems. In *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA'05)*. Springer, 201–213.

GUÉRET, C., MONMARCHÉ, N., AND SLIMANE, M. 2007. A biology-inspired model for the automatic dissemination of information in P2P networks. *Multiag. Grid Syst. 3*, 1, 87–104.

GUMMADI, K. P., DUNN, R. J., SAROIU, S., GRIBBLE, S. D., LEVY, H. M., AND ZAHORJAN, J. 2003. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. 314–329.

HARDER, T., HAUSTEIN, M. P., MATHIS, C., AND WAGNER, M. 2007. Node labeling schemes for dynamic XML documents reconsidered. *Data Knowl. Eng. 60*, 1, 126–149.

IAMNITCHI, A. AND FOSTER, I. 2004. *A Peer-to-Peer Approach to Resource Location in Grid Environments*. Kluwer Academic Publishers, Norwell, MA, 413–429.

JELASITY, M., MONTRESOR, A., AND BABAOGLU, O. 2009. T-man: Gossip-based fast overlay topology construction. *Comput. Netw. 53*, 2321–2339.

KLEINBERG, J. 2000. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC'00)*. 163–170.

KO, S. Y., GUPTA, I., AND JO, Y. 2008. A new class of nature-inspired algorithms for self-adaptive peer-to-peer computing. *ACM Trans. Autonom. Adaptive Syst. 3*, 3, 1–34.

MANKU, G. S., BAWA, M., AND RAGHAVAN, P. 2003. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th 2001 Conference on USENIX Symposium on Internet Technologies and Systems (USITS'03)*.

MAYMOUNKOV, P. AND MAZIÈRES, D. 2002. Kademlia: A peer-to-peer information system based on the XOR metric. In *Revised Papers from the 1st International Workshop on Peer-to-Peer Systems (IPTPS'01)*. Springer-Verlag, 53–65.

PANZIERI, F., BABAOGLU, Ö., FERRETTI, S., GHINI, V., AND MARZOLLA, M. 2011. Distributed computing in the 21st century: Some aspects of cloud computing. In *Dependable and Historic Computing*. Lecture Notes in Computer Science, vol. 6875. Springer, 393–412.

PEDERSEN, T. B. AND JENSEN, C. S. 2001. Multidimensional database technology. *IEEE Computer 34*, 40–46.

RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. 2001. A scalable content-addressable network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*. 161–172.

RODRIGUES, R. AND DRUSCHEL, P. 2010. Peer-to-peer systems. *Comm. ACM 53*, 72–82.

ROWSTRON, A. AND DRUSCHEL, P. 2001. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. Lecture Notes in Computer Science, vol. 2218, 329–350.

SCHMIDT, C. AND PARASHAR, M. 2004. Enabling flexible queries with guarantees in P2P systems. *IEEE Internet Comput. 8*, 3, 19–26.

STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*.

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

Received October 2011; revised March 2012, April 2012, May 2012; accepted June 2012