DDT: A Distributed Data Structure for the Support of P2P Range Query

Davide Carfí, Massimo Coppola, Domenico Laforenza
ISTI,CNR
Via Moruzzi, Pisa, Italy
Email: {davide.carfi,massimo.coppola,domenico.laforenza}@isti.cnr.it

Laura Ricci
Dipartimento di Informatica
Largo Bruno Pontecorvo, Pisa
Email: ricci@di.unipi.it

***Abstract*—This paper defines and evaluates a hierarchical distributed data structure, Distributed Digest Trie, supporting range queries in P2P systems. Providing efficient support for these queries is currently a challenging research issue in the P2P field, as classical approaches based on Distributed Hash Tables (DHT) are often not suitable for this kind of queries, due to the loss of locality introduced by the hashing function. Distributed Digest Trie exploits the DHT only to define a uniform assignment of logical identifiers to peers while each key is managed by the peer publishing it. Each peer is paired with the leaf of the trie corresponding to its logical identifier. An internal node of the trie stores a digest summarizing the keys published by the peers paired with the leaves of the tree rooted at that node. A proper mapping function is defined to map the internal nodes of the trie to the peers. The digests stored at the internal nodes are exploited to guide the search process for the resolution of the range query. Different aggregation techniques are proposed. A set of experimental results compare these techniques, evaluate the cost of dynamic updates of the data structure and the network traffic generated by the method.**

I. Introduction

The problem of *resource discovery* in massively distributed environments is currently an active research area [1]. A support for *resource discovery* is required, for instance, in Grids and in distributed environments supporting collaborative computing. Several recent proposals exploit the P2P computational model to define a scalable resource discovery support. Most of them are based on the *Distributed Hash Table* (DHT) approach, which offers a simple put/get API for defining a distributed directory service. Although several DHTs are currently available [2], [3], this technology is still developing the functionalities needed in order to support complex queries (range, multi-attribute or similarity queries) which are required by high level applications/services.

Several extensions of the basic DHT model for the support of *range queries* have been recently proposed [4]–[9]. Existing approached may be classified as follows. A first set of proposals [8] are based on the definition of a *locality preserving hashing function* which keeps the locality of the keys in order to support range queries. The main drawback of these proposals is that the load balancing properties of the DHT are no longer guaranteed, as we sacrifice the uniformly scattering of keys within the DHT. Other proposals [9] leverage a *space filling function* to define a linearization of the multi dimensional space of the resource attributes, so that the resulting space can be easily mapped to the underlying DHT. Finally, some approaches utilize the DHT as a communication substrate to build a hierarchical distributed data structure. They enhance the queries expressivity by maintenance a distributed index of resources.

We propose the *Distributed Digest Trie* (DDT) a distributed data structure which is based on a *trie* structure. The DDT improves the DHT query expressivity by providing the execution of range queries on a single attribute.

A *trie* is a data structure based on a hierarchical tree structure where each node corresponds to a distinct prefix of the domain value and all descendants nodes share a common prefix with the relative ancestors. DDT builds the distributed *trie* on the DHT substrate, it exploits the DHT to define a uniform assignment of peers identifiers. The alphabet of the trie is based on the $\{0, 1\}$ peer identifiers symbols. In this way we define a binary trie where each leaf represent a single peer of the DHT network. The internal nodes of the trie are partitioned among network peers by a mapping function. The mapping function takes into account several issues like the distribution of the trie, the workload or the reorganization problems. While most approaches presented in the literature are based on computationally expensive algorithms to keep the distributed data structure consistent, as peer join and leave the overlay, our proposal aims at keeping low the complexity of join/update operations. We show that the worst case complexity of DDT join and update operations are logarithmically bounded in the number of peers.

The DDT resources are kept by the peers, thus the queries require a distributed exploration of the trie. DDT introduces the concept of aggregation function in order to compute, on the internal nodes of the trie, a *digest* of the resources published in the lower levels. DDT defines several aggregate functions allowing the appropriate summarization of different kinds of resources. In this paper several aggregate functions are defined and compared with each other, to evaluate their trade off between accuracy and space requirements. At the best of our knowledge, our proposal is the first defining an aggregation trie in a P2P environment.

The paper is organized as follows. Sect.II presents the main proposals currently in the literature that relate to our work. Sect.III introduces the main characteristics of DDT, while Sect.IV and Sect.V respectively give a more accurate definition

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8356
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8356

of the *Digest Functions* and the *Mapping Functions* exploited by the DDT. The main algorithms defined to support DDT operations are discussed in Sect VI. Section VII presents a set of experimental results. Finally, Sect.VIII reports our conclusions and discusses future works.

II. Related Work

Nowadays structured P2P networks are a good choice for sharing resources over massively distributed environment. Several structured overlays [2], [3], [10] have recently been proposed which adopt and extend *Distributed Hash Table* (DHT) mechanisms to support scalable search of resources. Among the numerous new approaches (see e.g. [1]) we discuss here those more closely related to our work. In particular, the CONE approach [11] is described in Sect. IV. The main drawback of the mentioned systems is that, while they define a scalable support for *"exact-match"* queries, they do not offer a proper support for more complex ones, like *multi attribute range queries*. In this section we discuss the main proposals presented into the literature for the support of this kind of query and compare the proposals with respect to the following issues:

- *Balancing of the workload*: the load for storing and searching resources should be balanced among the peers;
- *Dynamic Resources*: the system should minimize the amount of updates in the case of resource modification.
- *Space Requirement*: each peer should reduce the size of the data structures required to store and retrieve the resources, e.g. the size of the routing tables.

BAlanced Tree Overlay Network, Baton [4], is a structured P2P system which supports the execution of unidimensional range queries. Baton maintains a distributed *B-tree*. Each peer of the network is paired with a range of resources corresponding to a node of the tree. When a new peer P joins *Baton*, the peer builds a routing table to store the links to other peers managing the tree. The routing tables are stabilized to guarantee that the tree remains balanced. The stabilization process is also required when a node changes the value of its resource. This approach guarantees both a good balance of the load and a that the execution of the *"range queries"* has a complexity which is *logarithmic* in the depth of the tree. The main drawback of the Baton approach is the stabilization process which is exploited when a new peer joins the network/updates its resource to guarantee that the distributed tree remains balanced. The overall performance can be degraded, due to this process, when the amount of updates is high. Our DDT proposal distinguishes from Baton as we explicitly aim at a good trade-off between the efficient management of resource dynamicity and the overhead of keeping the workload well balanced.

[7] presents a proposal based on a binary tree, the *Range Search Tree (RST)*. The RST builds a distributed tree over the peers in the overlay whose goal is to support range queries. Each leaf of the tree is paired with a single value and each internal node stores the union of ranges of values paired with its children. Each peer manages one or more nodes of the tree according to a *"Load Balancing Matrix", (LBM)*. The LBM keeps track of the queries and of the publication workload for a specific resource. The matrix can be retrieved from a special peer node called *"head node"*, or a discovery algorithm can be used to approximate it to avoid overloading the head node. When a peer joins the network it is paired with all the nodes belonging to a *Path* into RST, i.e. a set of RST nodes where the value published by the peer falls into. The concept key is to select only a subset of the nodes from the path. This set of nodes is called *"band"* and it is dynamic calculated in accordance with the range queries workload. The band is also stored as the LBM and is updated through the *"Path Maintenance Protocol"* (PMP). A RST search operation consists in retrieving the band of the published value, and afterwards in computing a decomposition of the node set of the range query into a set of RST nodes to query separately. The main drawbacks of the RST proposal are related to the LBM/band maintenance and to the resource dynamism management. In the first case, the overloading of the head node degrades the performance of the discovery. In addition the *Path Maintenance Protocol* increases the message traffic in the network. The last critical point is the overhead of continuous remove and registration operations in the presence of a high level of resources updates.

In the previous proposals the resource dynamism is one of main limitations. For this reason the current literature proposes further P2P approaches whose goal is to improve the management of dynamic resources. The *Range Category Tree (RCT)* [6] is a distributed data structure supporting the execution of multi attribute range queries and devised in the Grid computing context. *Range Category Tree (RCT)* exploits a balanced binary tree structure and organizes the resources on a *Primary Attribute (PA)* concept. A primary attribute is an attribute which best describes the characteristic of the resource and for each primary attribute a different RCT is built. Unlike the previous RST proposal in the RCT each nodes manages a specific range of PA and the range value distribution has a *"load-aware self-adaptation"*. The authors show how to enable the resource discovery across the different RCTs through a RCT Index Service (RIS) network as an upper layer. The RISs can support a service retrieval like a UDDI registry.

Finally, The *Tree Vector Indexes*, (TVI) [5] is P2P system that allows to execute range query and manages efficiently the resources dynamism. TVI uses an index to route the query toward the areas of the network where matches can be found. The peers are connected through an *undirected spanning tree* and each connection link is paired with a data structure that is a *bit vector*. The bit vector drives the query process and it can be efficiently updated when a resource is modified. The TVI can represent an excellent strategy for dynamic content retrieval on P2P networks, but its performance evaluation is not yet clearly related to the topological characteristics of the network overlay.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8356
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8356

III. Distributed Digest Trie: General Definitions

The DDT exploits a uniform *SHA1 Hashing Function* to assign h-bits identifiers defined in the logical space $S = \{0 \dots 2^{h-1}\}$ to the peers joining the structure. A trie over the alphabet $A = \{0, 1\}$ of the identifiers is defined so that each node of the trie corresponds to a prefix of the identifiers defined in the logical space. A *Distributed Hash Table* is exploited both to assign the identifiers of S to the peers and to support their bootstrap on the overlay. Each peer is paired with the leaf of the trie corresponding to its identifier.

In *DDT* each key is stored by the peer which publishes it. This distinguishes *DDT* from the classical DHT approach, where the key published by a peer may be mapped by *SHA1 function* to any peer through a specific DHT mapping function.

In our approach, the leaves of the trie store the data published by the peers, while each internal node stores a *digest* summarizing the information stored at the leaves of the subtree rooted at that node. The main purpose of the *digest* is to guide the search of data satisfying the range queries submitted by the peers. Each query is propagated bottom up starting from the node submitting the query and the information stored at the internal nodes is exploited to decide if a subtree may include values matching the range defined by the query.

It is worth noticing that the update of a key in a leaf of the trie may require updating the *digest* information in a subset of the nodes on the path from that leaf to the root of the trie. The definition of a proper *Aggregation Function* should balance the level of approximation introduced by the digest with the number of updates required when a key is modified. It is worth noticing that, in any case, the number of updates is bounded by the height of the trie. Section IV will discuss a set of aggregation functions characterized by different levels of approximation.

Each node n of the trie is assigned to a peer p by a proper *Mapping Function*. The following definition introduces a family of Mapping Functions which enables a straightforward definition of the most important operations of DDT.

$$Map(n) = \begin{cases} p : \text{SHA1}(p) = id \wedge ID(n) = id \\ \qquad\qquad \textbf{if } Is\_Leaf(n) \\ p \in \{p' : \text{SHA1}(p') = id, \\ \qquad ID(n) = id', pre(id', id)\} \\ \qquad\qquad \textbf{if } \neg Is\_Leaf(n) \end{cases}$$

In the formula, SHA1 is the hash function mapping a peer to the logical space of the identifiers, and the remaining notation is defined in table I, together with further definitions used in the following.

According to the previous definition, each node of the trie is mapped to a single peer while a peer possibly manages a set of logical nodes of the trie. While the mapping of the leaves to the peers is defined by their logical identifiers, the mapping function chooses from a set of possible candidates the peer to be paired with an internal node of the DDT. The only restraint introduced by our definition of mapping is that, given an internal node n, $ID(n)$ is a prefix of the identifier of the peer paired with n. This restraint has been introduced to make the bootstrap of the peers easier. As a matter of fact, the previous definition guarantees that a peer p may choose as bootstrap peer the one sharing the longest common prefix with itself, and receive from this peer the information required to correctly join the *Digest Trie*. As shown in Sect. VI, the bootstrap peer can be detected by exploiting the DHT routing.

Symbol	Meaning
p, q	*peer of the DHT network*
n	*node of the DDT trie*
id	*peer identifier*
h	*height of DDT trie*
k	*resource compressor factor*
m	*number of resource to summarize*

Function	Meaning
ID(n)	*prefix assigned to n by the trie*
pre(id_1, id_2)	*predicate true iff id_1 is a prefix of id_2*
Is_Leaf(n)	*predicate true iff n is a leaf*
Key(p)	*key published by p*
Leaf(p)	*the leaf l of the trie such that ID(l)=SHA1(p)*
Children(n)	*the set of children nodes of node n*
Level(n)	*level of n in the DDT*
Is_Leaf(n)	*predicate true iff n is a leaf*

TABLE I
Symbols and Functions

Different mapping functions will be introduced in Section V each one taking into account different issues, like the balance of the load among the nodes, the efficiency of the search process and so on.

Each peer p stores, besides the key it publishes, the set of digests paired with the logical nodes assigned to it and a *routing table* whose structure is determined by the chosen mapping. The routing table stores the *IP Addresses* of the peers, instead of their logical identifiers. Each message can be directly sent to its destination peer, saving the cost of DHT routing. The *overlay network* connecting the peers is defined by the links stored in the routing tables of the peers.

In DDT, the trie is visited *bottom up* starting from the leaf of the DDT corresponding to the peer which has submitted the query. Even if different strategies may be exploited to realize a distributed visit of DDT, the main goal of our approach is to avoid that the nodes at the upper levels of the trie receive most queries so becoming a bottleneck of the system.

For this reason, when a peer receives a query it first propagates the query to the subtrees rooted at the nodes mapped to itself, before propagating the query to the upper levels of the trie, The subtrees are chosen according to the estimation of the matches which may be found for the query in that subtree. This estimate is evaluated through the digest associated to the root of a subtree. The accuracy of the digest has a great impact on the efficiency of the search process.

Different strategies may be exploited to visit the subtrees. For instance, subtrees may be ordered according to the matches and then visited sequentially or a subset of the promising peers may be visited in parallel. A proper balance between

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8356
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8356

the degree of parallelism and the amount of traffic generated for a query should be properly defined.

An important issue is also the definition of a proper criterion to decide when the propagation of the query should be stopped. An approach based on the definition of a TTL for the query, like the one exploited in Gnutella, is not suitable in our case. Approaches based on the backward propagation of the query matches should be avoided as well, because of the high level of the generated traffic. Our approach will be described in more details in Section VI

IV. The Aggregation Functions

This section presents several aggregation functions characterized by different degrees of approximation and of computational complexity. Since the main goal of $Cone$ [11] is to define a distributed support for queries such as "find k resources whose value is $> x$", a *distributed heap* is defined where the *maximum* function is exploited to aggregate the values stored in each subtree. In this way, the maximum value is always stored at the root of the tree and a query may be solved by visiting the aggregation tree up to a node storing a value greater than x. Further values matching the query may be found on the way from this node to the root.

The main advantage of this approach is its low computational complexity. Furthermore, a simple mapping of the internal nodes of the aggregation tree to the peers may be defined by assigning a node to the peer storing the aggregated value. This is possible because the aggregate value always equals one of the key published by the peers.

On the other way round, the $Cone$ aggregation strategy is not suitable for to support range queries. As a matter of fact, a subtree may be cut off only if the maximum value stored at its root is lower than the lower bound of the query, while it returns no significative information if at least one key is smaller than the maximum.

Bivectors have been exploited to implement *routing indexes* [5] for unstructured networks. A bitvector is defined by selecting $k+1$ division points within the interval $[l,u]$ of keys values, $l = p_0 < p_1 < \ldots < p_k = u$ such that $[l,u]$ is partitioned into k distinct intervals $[p_i, p_{i+1}), i = 0, 1, \ldots k-1$. The digest summarizing the keys stored in a subtree S is defined by the bitvector $B = (b_0, b_1, ..., b_k)$ such that $b_i = 1$ if and only if exists a key in S belonging to $[p_i, p_{i+1})$.

The main advantage of this approach is the straightforward implementation of the merge of a set of bitvectors which can be computed by considering their bitwise disjunction. This operation should be computed at each internal node n of the *Digest trie* to define the bitvector associated with n as a function of those associated to its child nodes.

On the other way round, the approximation introduced by a bitvector may result too coarse to support the resolution of a range query. As a matter of fact, a bitvector shows if at least one key belongs to one of its intervals but it is not able to return the *number of keys* included in that interval. Furthermore, the approximation becomes less accurate as the number of aggregations step increases, i.e. at the upper levels of the *Digest Trie*. Some information may be recovered when the key distribution is known in advance by defining a partition of the key space such that intervals are more narrow where keys are more frequent.

The *Q_Digest, Quantile Digest* approach [12] improves the approximation accuracy of bitvectors by considering intervals of different sizes and by pairing a counter with each interval, defining the number of keys belonging to the interval.

A *Q_Digest* $Q = \{([l_1,u_1], count_1), \ldots ([l_i,u_i], count_i), \ldots, ([l_t,u_t], count_t)\}$ is a structure including a set of different sized intervals, or *buckets*, where each interval $[l_i, u_i]$ is paired with a counter $count_i$. The counter defines the number of keys belonging to that interval. For the sake of simplicity, in the following we will suppose that the size of the interval of the key values is a power of 2. The intervals are defined by considering a *binary partition* of the interval of the key values which is represented by a binary tree. Each leaf of the tree corresponds to a single value interval, while the size of the intervals associated to internal nodes increases as their level in the tree. The interval covering the whole key space is associated with the root of the tree. The depth of the tree is $log(n)$, where n is the dimension of the key space.

A *Q_digest* includes a subset of the intervals defined by the binary partition. The cardinality of this subset depends upon a *compression parameter* k which may be tuned to balance the complexity of computing the *Q_digest*, r.s. its storage requirements, versus the accuracy of the returned approximation.

[11] suggests to exploit two properties, the *Digest Properties*, in order to define the intervals which should be inserted in a *Q_Digest*. The first one affirms two adjacent intervals with low counters should be merged. This corresponds to merging two children of the binary partition tree into their parent node provided that the counter resulting from adding their counter and that of the parent node is under a given threshold. In this way intervals with a too low value are not inserted in the *Q_Digest*. The second property asserts that an interval should not have a too high counter, unless it is a leaf.

If m is the number of keys, and k is the compression parameter, previous conditions may be defined as follows:

$$count(n) \leq m/k$$
$$count(n) + count(n_p) + count(n_s) > m/k$$

where n is a node of the binary tree associated with the $QDigest$, n_p, rs. n_s is the parent, r.s the sibling of n in the tree.

The compression algorithm applies the digest properties by visiting the binary tree bottom-up, starting from its leaves. The merge of a pair of *Q_Digest* Q_1 and Q_2 defined on the same value space is computed by defining a *Q_digest* Q on the same value space such that each node of Q is first paired with a counter obtained by adding the counters of the corresponding intervals in Q_1, Q_2, then by applying the compression algorithm to it.

Figure 1 shows a Q_Digest defined on a value space [1..8] where the number of values in this space is 18. The leaves of

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8356
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8356

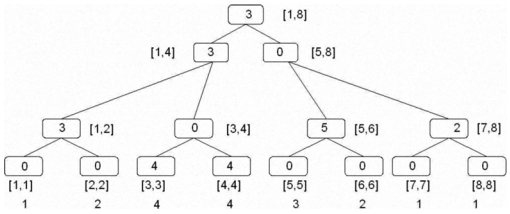

Fig. 1. A Quantile Digest

the trie corresponds to the intervals including a single value and the counts under the leaves shows the number of keys with that value. These are the initial values for the counters of the leaves, while the counters of the internal nodes of the tree are initially set to 0. The *Q_digest* is obtained by applying the compression procedure to these initial configuration of the tree. Figure 1 shows the tree resulting from the compression where the compression parameter is 3 and the value of m/k is 6.

The counter shown inside the nodes are resulting from the compression. The intervals with a counter equal to 0 are not included in the *Q_Digest*. Note that the intervals $[1,1]$ and $[2,2]$ have been merged because they do not satisfy the first Digest property. The resulting counter floats till the root because the merge is recursively applied to the ancestors nodes. The intervals $[3,3]$ and $[4,4]$ are not merged because they satisfy the first and the second Digest property. Finally, intervals $[5,5]$ and $[6,6]$ are merged into their parent node and the same applies to intervals $[7,7]$ and $[8,8]$, but the merge process stops at their parent nodes, because the first condition is satisfied at these nodes.

[11] proves that a *Q_Digest* constructed with compression parameter k has size at most $3k$ and that in a *Q_Digest* Q created using a compression parameter factor k, the maximum error in count of any node is $\frac{\log(\sigma)}{k}$, where σ is the size of the size of the interval of the value keys and m is the number of keys.

Let us now consider a range query $[l,u]$ and a $Q = \{([l_1,u_1],count_1),\ldots([l_i,u_i],count_i),\ \ldots,([l_t,u_t],count_t)\}$ An approximation of the number of matches of the range query may be obtained by as follows:

$$\sum count_i : [l_i,u_i] \bigcap [l,u] \neq 0$$

Finally it is worth noticing that each compression step introduces a degree of approximation since it merges two intervals into a larger one. The corresponding counter exactly defines the number of values included in that interval, but it cannot describe the distribution of the values within the interval. On the other way round, the information returned by the counter is more accurate of that contained in the bitvector and can be exploited to decide more accurately if a subtree should be visited.

V. The Mapping Functions

According to the prefix condition introduced in the Section III, each internal node n of the DDT may be paired with any peer whose identifier is prefixed by $ID(n)$. Distinct mappings correspond to different choices of these peers.

Among these mappings we will consider those where each peer manages a *single sequence of nodes* from its leaf to an ancestor node on the path to the root of the DDT. These mappings will be referred in the following as *single segment mappings*.

These mappings present several advantages. First of all, the number of hops between different peers during the query resolution process and, as a consequence, the overall network traffic, is reduced. As we will see in the following section, both the complexity of the join operation and of the routing tables is reduced as well.

$$Map(\{n_1,n_2\}) = Map(\{n_1\}) \cup Map(\{n_2\})$$
$$Map(\{n\}) = \begin{cases} p \text{ if } leaf(n) \wedge ID(n) = \text{SHA1}(p) \\ q \text{ if } (\neg leaf(n) \wedge (Map(Children(n))) = q) \vee \\ \quad \vee (\neg leaf(n) \wedge \\ \quad \wedge Map(Children(n)) = \{q,p\} \wedge \\ \quad \wedge\ \ Choice(p,q,n) = q) \end{cases} \quad (1)$$

Eq. (1) shows a general definition of a mapping satisfying the *single segment* property. The function maps each leaf of the DDT to the peer whose SHA1 identifier equals the leaf identifier. Each internal node n with a single child n', is assigned to the peer managing n'. Otherwise, the *Choice Function* maps n to one of the peers paired with its children. Distinct definitions of *Choice* define different mappings.

Let us consider the segment of nodes assigned to a peer p by a single segment mapping. The parent of each node assigned to p, apart from the upper level node, is paired with p itself. If the upper level node is different from the root, the peer q managing the parent of the upper level node will be referred in the following as *father peer* of p and p will be referred as *child peer* of q. Note that, while in the general case a peer may have a set of father peer, the single segment mapping defines at most a father peer for each peer of the DDT.

[11] defines a single segment mapping where each internal node n is assigned to the peer storing the *maximum key* choosen among the peers paired with the children of n. This mapping is guided by the digest information stored at the nodes of the DDT:

The corresponding *Choice function* is shown in the Eq.2.

$$Choice(p,q,n) = \begin{cases} p & if \quad key(p) \geq key(q) \\ q & otherwise \end{cases} \quad (2)$$

In this case the third parameter, i.e. the reference to the logical node which has to be mapped, is not exploited. It is worth noticing that while this approach is straightforward in [11] because the maximum function returns one of the keys paired with the peers, the choice is more complex when digest like the *Bitvector* or the *QDigest* are exploited.

Fig. 2 show an example of this mapping. The value shown inside each node is the digest paired with that node, while the peer managing the node is defined close to the node. Note

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8356
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8356

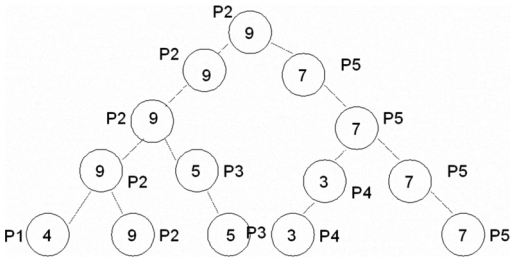

Fig. 2. A Digest Based Mapping

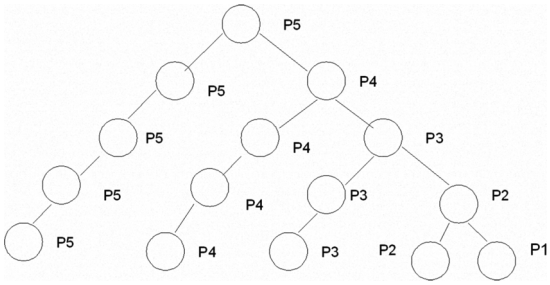

Fig. 3. A Load Balancing Mapping

that each peer $P_i, i = 1, .., 5$ manages a sequence of nodes starting from its leaf up to an ancestor node. Each internal node is assigned to the peer publishing the key corresponding to the digest paired with that node. Note also that the peer P_2 manages each node on the path form its leaf to the root, because it publishes the maximum key.

The main drawback of this approach is that the resulting mapping generally *does not balance* the load among the peers. For instance, in the mapping of Fig. 2, the peer publishing the maximum key manages any node on the path from its leaf to the root of the trie.

The definition of a mapping taking load balancing into account is a true challenge, because it should balance the complexity of the mapping function with the benefits introduced by a proper load balancing.

First of all, a proper definition of the load of a peer is needed. A straightforward solution is to define the *load* of a peer p as the amount of DDT nodes assigned to it. Nevertheless, this approach does not take into account that some nodes assigned to p may have a single child managed by p itself. Instead, we will consider a more refined definition taking into account *the number of child peers of p*, since this value approximates the amount of queries which p should manage.

A simple heuristics to define a mapping balancing the load is defined as follows

$$Load(p,l) = |\{n_k : n_k \in Path(leaf(ID(p),l) \wedge \wedge Sibling(n_k) \neq \perp\}|$$

$$Choice(p,q,n) = \begin{cases} p & \textbf{if } \; l = level(n) \; \wedge \\ & \quad Load(p, l-1) \leq Load(q, l-1) \\ q & otherwise \end{cases}$$

Given a peer p, let us consider the path from $Leaf(p)$ to its ancestor of level l and suppose that each node on this path has been assigned to p. The load of a p on this path, *Load(p,l)*, is defined as the cardinality of the set of all the nodes on the path which have a sibling node in the DDT. It is worth noticing that each sibling, if existent, is surely paired with a peer different from p. The function *Choice(p,q,n)*, given the peers p and q and a node n at level l belonging to the path from $Leaf(p)$, r.s. $Leaf(q)$ to the root selects between them the peer with the lowest load at level $l-1$.

Fig.3 shows an example of this mapping. Note, for instance that the *Choice Function* chooses peer P_3, at level 2 between P_2 P_3, because its load is 0, while the load of P_2 is 1. Furthermore, the load of P_5 is 1 even if P_5 manages all the nodes form its leaf to the root since each node on this path has a single child which is paired with P_5 itself.

A routing table corresponding to a single segment mapping has at most h entries, where h is the height of the DDT and presents the following structure. Each entry of the routing table of a peer p corresponds to a node n of the DDT and stores the digest paired with that node and may store a pointer to a child peer of p if a child of n is managed by a peer different from p. The digest paired with a node summarizes the information stored in the subtree rooted at that node.

VI. The DDT Operations

This section describes the main operations supported by *Digest Trie*, i.e. *join*, *leave*, *find* and *update key*. The first two are exploited when a peer joins, r.s. leaves the DDT. The search of k values belonging to a given range is defined by the find operation, while *change key* updates the digests paired with the nodes of the trie when a peer modifies its key. Each operation will be described in more details in the following sections.

A. Join

Before joining the DDT, each peer gets a *logical identifier* which is assigned to it by exploiting the *SHA1 hash function* of the underlying DHT. In this way p logically joins the DDT at the leaf l of the DDT such as SHA1$(p) = ID(l)$.

The joining process requires two different steps. The first one detects the segment of internal nodes paired by the mapping function to the joining peer p. The second one updates the digest information on the path from $Leaf(p)$ to the root by considering $key(p)$.

In the first step, p first detects the *maximum prefix* MP shared between its identifier and those of the peers belonging to the DDT, afterwards the *least common ancestor, LCA*, that is the node of the DDT identified by MP is detected. Note that more than one peer may share MP with p and that one of these peers manages the LCA. Any node on the path from $Leaf(p)$ to LCA is managed by p. p than should contact the peer q paired with the LCA to decide which nodes on the path from LCA to the root should be managed by itself. Note

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8356
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8356

that q acts as a *bootstrap peer* for p. p contacts q and they apply the Choice function defined by the mapping to decide which of them should manage the common segment of internal nodes. If p is choosen by the choice function, the process is recursively repeated by considering the father peer of q until the choice function chooses a peer different from p. During this process p builds its routing table and any peer involved in the process updates its routing table.

The underlying DHT is exploited to detect the bootstrap peer. As shown in [11], one of the peers sharing the maximum prefix with p is definitely either *the successor* or *the predecessor* of p in the DHT. Note that the predecessor/successor of p in the DHT may be detected by exploiting the routing tables of the DHT which are built at the underlying level when the peers join the DHT. Hence p contacts its successor(or its predecessor) in the DHT. If this peer is not paired with the LCA it forwards the request of p to its parent and the forwarding is carried on until the peer q paired with the LCA is found.

The final step of the join requires the update of the digests paired with the nodes on the path from $Leaf(p)$ to the root of the DDT. First p updates the digest of the nodes assigned to it by considering its key, starting from $Leaf(p)$ and going back up the tree until the new digest differs from the previous one. If the update reaches its upper level node, p notifies the digest associated to this node to its father peer and the update procedure is recursively applied by the ancestors of p until the updated digest equals the old one.

It is worth noticing that the level reached by the update procedure increases as the accuracy of the digest. On the other hand, a more accurate digest improves the search process, by decreasing the number of nodes visited to satisfy the query.

Operation Analysis: the first step of the join operation involves localizing the LCA. This operation is performed through the DHT substrate with a worst-case of $O(\log N)$ exchanged messages with other peers. The second step of the join requires checking of the mapping function and of the digest update. Here too we at most contact all the peers along the path to the root node. The maximum number of contacted peers is thus $h = \log N$, and the overall worst case complexity of the join operation is $O(\log N)$.

B. Find

We consider queries like $Find(l \leq x \leq u, k)$, where l, r.s. u is the lower, r.s. the upper bound of the range defined by the query and k is the number of required values. A query q submitted by a peer p, i.e.the query peer, is propagated *bottom up* in the DDT, starting from the leaf corresponding to p. The *Find* operation exploits a fully distributed algorithm where no central entity coordinating the search of query matches does exist. As a consequence, each node receiving the query should autonomously decide when the visit has to be stopped. It is worth noticing that each node whose key k matches the query directly sends k to the query peer p. This avoid the definition of a *backtracking mechanism* which should considerably increase the traffic on the overlay. On the other way round, each node should autonomously decide if the query should be stopped, without receiving the result of the match process form its neighbours.

Let us suppose that q reaches a peer t which is paired with a segment s of nodes spanning from level l to level $l+k$ of the DDT. t first order the subtrees rooted at the nodes belonging to s according to some strategy, then exploits the digest paired with the subtrees to estimate the number of matches for the query which may be found in each subtree.

The visit of the DDT stops when the number of estimated matches exceeds k. It is worth noticing that t propagates the query to its father peer if and only if the number of estimated matches in its subtrees is $\leq k$. Note that the goal of this strategy is to propagate the query to the upper levels of the DDT only when this is really necessary thus avoiding that the upper levels of the tree become a bottleneck.

Furthermore, a proper order of the subtrees rooted at the nodes in s should be defined in order to minimize the number of visited nodes and, as a consequence, the network traffic. This ordering should take into account both the number of nodes of the subtree and the number of matches for the query.

Finally, it is worth noticing that the level of approximation introduced by the digest may greatly affect the search process. As a matter of fact an under estimate implies that the query is propagated to the upper level of the trees even when this shouldn't be necessary, thus creating a bottleneck at the upper levels of the tree. On the other hand, a over estimate may stop the search too early, before k matches are really found. To face the latter problem, the peer stopping SP the search process sends to the query peer QP a *stop message* which notifies to QP the point of the DDT where the search could be resumed. QP may later resume the search process, if the number of matches for the query received is $\leq k$. Note that a proper algorithm should be defined in this case, because the structure of the DDmay be changed due to the dinamicity in the meanwhile.

Algorithm 1 shows the pseudo code of the *Find*.

Operation Analysis: the find operation involves a distributed exploration of the trie. The exploration process executes a leaf-to-root visit, at each level starting an exploration of potentially interesting sub-tries. The overall number of explored peers depends on the digest information at the nodes. The more accurate the estimate provided by the aggregation function, the lesser is the amount of peers explored. In the worst case a DDT find requires $O(N)$ messages.

C. Leave

When a DDT peer leaves the overlay voluntarely, it first invokes the *leave operation*. This operation transfers the local Routing Table of the leaving peer to its parent peer which integrates the received information with its routing table and propagates to its children the update. Otherwise, if the peer leaves unexpectedly the network due to an unexpected crash, the peers which are connected to it in the DDT overlay detect the failure and they exploit the underlaying DHT to fix the DDT consistency. Each peer performs a join procedure

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8356
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8356

Algorithm 1 Find

```
procedure N.Find(Q, k, QueryNode, Dir)
    if Matches(Q, LocalKey) then
        send(QueryNode, N.Key)
        k ← k − 1
        if k = 0 then
            exit
    for all Son ∈ Sons do
        Ext(Son) ← Q ∩ Digest(Son)
    if (dir =' down') then
        for (i = 1, Length(Sons)) do
            if Ext(Son(i)) ≠ 0 then
                send (Sons(i), Q, k, QueryNode,' down')
    else
        SSons ← Sort ({Sons : Ext(Son) ≠ 0})
        for i = 1, |Sons| do
            send (SSons(i), Q, k, QueryNode,' down')
            k = k − Ext(i);
            if k < 0 then
                send (QueryNode,' SearchStop')
                exit
        if (k > 0 and ¬N.isRoot()) then
            send (ParentNode, Q, k, QueryNode,' up')
```

to localize its *least common ancestor* in the network thus updating the mapping and the digest functions. The detailed description of the leave operation may be found in [13].

Operation Analysis: if we consider a volunteer leave, the cost analysis is the same as the join operation. DDT checks the mapping function to verify the structure consistency and executes the digest update, and at most $O(\log N)$ messages are exchanged among peers. Otherwise, if a peer unexpectedly leaves the network, the trie can get disconnected. The DHT substrate is used to localizes the LCA peers needed to reconnect the trie (they are at most $\log N$), each one is found using at most $O(\log N)$ messages. Trie consistency can then be restored with a function mapping checking. The overall worst-case complexity of the leave operation is thus $O(\log N^2)$.

D. Change Key

When a peer changes the value of its key, the digests paired with a subset of the peers on the path from its leaf to the root may be updated. The procedure is similar to the join procedure performed when a new peer enters the overlay, which has been discussed in Sect. VI-A.

Operation Analysis: a resource change involves in the worst case both a digest update and a mapping reorganization. As for the join, a path-visit toward the root of the distributed trie is needed. Therefore the worst-case complexity of the change is $O(\log N)$ messages.

VII. Experimental Results

This section describes and evaluates a prototype of *DDT* developed through the *OverlayWeaver toolkit, OW* [14].

OW is an overlay construction toolkit which provides a common high level API to develop distributed services. The architecture of the toolkit is decomposed into multiple components, the routing driver, the routing algorithm, the messaging and the directory service. It provides multiple routing algorithms for different $DHTs$ and enables a large-scale emulation with a fair comparison between algorithms The high level functionalities defined by *OW* facilitates both the implementation and the evaluation through an emulation of *DDT*.

The main goals of the experiments has been the evaluation of the following aspects of the DDT:

- Analysis of the Digest Functions
- Evaluation of the Query Load on the Root
- Analysis of the System in presence of Data Dinamicity

The test bed is based on 1,000 virtual nodes of an emulated network. The data distribution among the peers follows a standard *Zipf law*, where the keys domain is $[0, 100)$ and the density of the keys is maximal in the left part of the key domain.

The same Zipf distribution has been exploited to generate the queries. A query **k:[a, b]** is defined by the following parameters:

- **k** the minimum number of resources required by the query;
- **a**, rs. , **b** is the lower, r.s. the upper bound of query;

Each query **k:[a,b]** is generated according to the following schema:

- a is generated by the inverted standard Zipf law in the domain $[0, 100)$
- $b = a + c$ where c is uniformly selected in the domain $(a, 100)$
- $k = \alpha \times [F(b) - F(a)]$, where F is the Zipf CDF and $\alpha \in \{1, \frac{1}{2}, \frac{1}{4}\}$

A. Analysis of the Digest Functions

This test case analyzes the different digest exploited by DDT. We remark that the digest function has an important role in the discovery process to prune useless branches of the tree. The test is executed as follows. 100 nodes have been randomly selected, afterward each node submits a query generated according to the pattern previously described. At the end of search process we compute, for each query, the numbers of peers which have been involved in the query resolution and the number of matches collected for the query. This experiment has been executed for $\alpha = 1, \frac{1}{2}, \frac{1}{4}$. The effectiveness of the *Digest* may be measured by considering the average number of peers involved in the query resolution vs. the number of collected results.

The results of this test are shown in Fig. 4 and in Fig. 5. In Fig.4 the X axis corresponds to the α parameter which determines the minimum number of resources required by the query, while the Y axis reports, for each value of α, the average number of peers which are involved in the search process. The results are reported with respect to the Digest Functions introduced in the previous sections, i.e. the Maximum, the Bitvector and the Q-Digest. The Bitvector is defined by selecting 10 subintervals in the key domain $[0, 100)$ such that the subdivision takes into account the Zipf

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8356
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8356

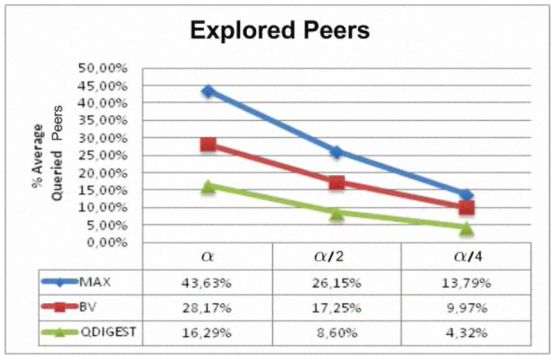

Fig. 4. Aggregate Functions: number of peers involved in the search process

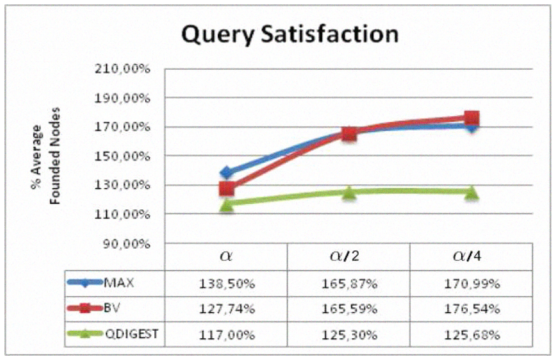

Fig. 5. Aggregate Functions: Average of query satisfaction

distribution of the keys (interval division points at 40, 50, 60, 70, 80, 85, 90, 93, 96, 100). Note that this approach requires a priori knowledge of the distribution of the keys. As far ad concerns the Q-Digest, the only parameter to fix is the compress factor whose value is 100. Note that in this case the definition of the digest does not require the knowledge of the resource distribution function. The experiment show the relevant impact of the aggregation function on the cost of the DDT find operation. The find with a Q-Digest function explores approximately 16 peers out of 1000. Although a very preliminary result, the experimental complexity looks much lower than our worst-case expectation $O(N)$.

The second test analyzes the query satisfaction, i.e. the average number of matches retrieved for each query, in the same test scenario. As a matter of fact, a digest may underestimate the number of matches for the query that are present in a subtree. when this happens, the number of matches returned to the query node is larger than the requested k. In our experiments, the query satisfaction exceeds 100% for all the digest functions, see Figure 5. This means that all the digest functions we studied exhibit some degree of underestimation.

On the other way round, the Q-Digest although exceeding k matches, exhibits about 20% less matches than the other digests. Thus Q-digest improves the ratio between peers involved in the search process and the number of retrieved matches. Q-Digest improves the search operation by decreasing the amount

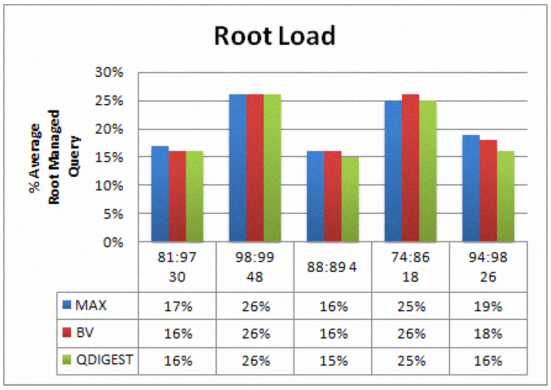

Fig. 6. Control Traffic: traffic load of DDT root peer

of involved peers, and by increasing the relative amount of retrieved resources. We can observe that Q-digest, with respect to the other digest functions, for all values of α answers the queries contacting at least 50% less nodes. We can conclude that the a self-adapting characteristics of the Q-Digest with respect to the Bitvector returns better results in the search process.

B. Analysis of Query Load on the Root

One of the most critical points of the DTT proposal is the impact of the traffic generated by the query resolution on the peer which manages the root node. In order to investigate this aspect we have analyzed the average number of queries which reach the peer managing the root of the DDT, for different query configuration and different digests. The test configuration is the same defined in the previous section. We have selected 100 nodes at random, each node submits the 5 queries which are shown in the histogram shown in Fig. 6. The queries are generated with the same distribution defined previously and with $\alpha = \frac{1}{4}$. The goal of the experiment is to analyze, for each digest function, the amount of queries which reach the peer which manages the root of the DDT. The histogram shows, for each query which is submitted by the 100 nodes, the percentage of the queries which reaches the peer managing the root.

The test shows that, for all digest, the amount of query load on the peer which manages the root is never larger than 26% with an average of 20%. This result confirms that even in the case where the approximation of the digest may result too coarse, i.e. the Maximum or Bitvector functions, the load of root node is widely acceptable.

C. Analysis of Dinamicity

The last test analyzes the impact of dynamic due to key updates in the DDT. As we have described in section VI the update of a key generally requires the propagation of the new digest information by triggering the update of the digests paired with a subset of the DDT nodes.

The test analyzes the behavior of DTT under frequent updates. The environment configuration is the same of the

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8356
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8356

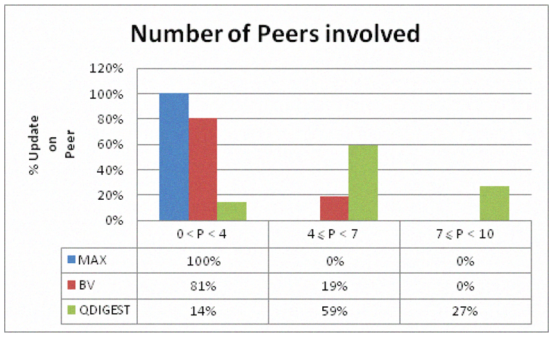

Fig. 7. Analysis of Key Updates

previous section, in addition we have generated 100 key updates where the updated values are generated by following the same Zipf-law distribution shown in the previous section. Each update is submitted by one peer which is selected randomly. The test analysis the number of peers involved into stabilization process. The emulation considers 1,000 peers and as a consequence the maximum height of DDT is 10. The behavior of the three digest functions presented in the previous sections is analyzed.

The histogram presented in Fig. 7 reports the number of updates which involves a number of peers defined by a set of ranges. For instance, the 80% of the updates involve between 1 and 3 peers when the Bitvector is exploited. It is worth noticing the trade off between the degree of approximation introduced by the digest and the behavior of the update operation. Even if the Max and Bitvector digest nail the updates with the average of 90% under 4 peers, the Q-Digest presents an acceptable behaviour in presence of keys updates (73% under 7 peers about which the 14% under 4 peers). On the other way round, the number of updates required by the Q-Digest is inevitably larger than those required by other digests because its accuracy is larger. As a conclusion, we can confirm that even if the performance of the Q-Digest is worse with respect to the others digest when the updates are considered, it represents the best compromise when considering the performance of the search vs. the update operations.

VIII. Conclusions

This paper proposes the Distributed Digest Trie, a hierarchical distributed data structure supporting range queries in P2P systems. The DDT structure is built on the top of a DHT. The DDT exploits both a *Mapping Function* to map the nodes of the logical tree to the peers and a set of *Digest Functions* to aggregate the keys published by them. We have defined and evaluated three different Digest Functions, namely Maximum, Bitvector and Q-Digest. DDT supports the execution of range queries by exploiting the digest information to drive the search process only toward those peers where matches can be found. The experimental results confirm the effectiveness of our approach. As far as it concerns the ratio between the number of nodes visited by the search process and those updated when a key is modified, the Q-Digest function gives the better trade-off so far between query efficiency and update overhead. A thorough understanding of the trade-offs implied in the choice of the mapping and digest functions will require further study. We are also currently extending the DDT to support *multi-attribute range queries* by exploiting a space filling based approach [9] to define a linearization of the multi attribute key space. The resulting space is then mapped to the DDT by exploiting the techniques proposed in this paper.

Acknowledgment

The authors acknowledge the support of XtreemOS, Project FP6-033576, Building and Promoting a Linux-based Operating System to Support Virtual Organizations for Next Generation Grids (2006-2010).

References

[1] R.Ranjan, A.Harwood, and R.Buyya, "Peer-to-Peer Based Resource Discovery in Global Grids: A Tutorial," *IEEE Communications Surveys and Tutorials*, vol. 10, no. 2, pp. 6–33, 2008.

[2] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, F. M. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 1, pp. 17–32, February 2003.

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

[4] H. V. Jagadish, B. C. Ooi, and Q. H. Vu, "Baton: a balanced tree structure for peer-to-peer networks," in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, 2005, pp. 661–672.

[5] M. Marzolla, M. Mordacchini, and S. Orlando, "Tree vector indexes: efficient range queries for dynamic content on peer-to-peer networks," in *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*, 2006, pp. 8 pp.+.

[6] H. Sun, J. Huai, Y. Liu, and R. Buyya, "Rct: A distributed tree for supporting efficient range and multi-attribute queries in grid computing," *Future Gener. Comput. Syst.*, vol. 24, no. 7, pp. 631–643, 2008.

[7] J. Gao and P. Steenkiste, "An adaptive protocol for efficient support of range queries in dht-based systems," in *Proceedings of the 12th IEEE international conference on network protocols (ICNP'04)*, 2004, pp. 239–250.

[8] M. Cai, M. Frank, J. Chen, and P. Szekely, "Maan: a multi-attribute addressable network for grid information services," in *Grid Computing, 2003. Proceedings. Fourth International Workshop on*, 2003, pp. 184–191.

[9] C. Schmidt and M. Parashar, "Enabling flexible queries with guarantees in P2P systems," *Internet Computing, IEEE*, vol. 8, no. 3, pp. 19–26, May 2004.

[10] P. Maymounkov and D. Maziãres, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[11] G. Varghese and G. M. Bhagwan, P.and Voelker, "Cone: Augmenting dhts to support distributed resource discovery," in *19th ACM Symposium on Operating Systems Principles, SOSP poster session*, 2003.

[12] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, "Medians and beyond: new aggregation techniques for sensor networks," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2004, pp. 239–249.

[13] D. Carfi, "Xcone: Range query in sistemi p2p," Master's thesis, University of Pisa, December 2008.

[14] K. Shudo, Y. Tanaka, and S. Sekiguchi, "Overlay weaver: An overlay construction toolkit," *Computer Communications*, vol. 31, no. 2, pp. 402–412, February 2008, framework available at http://overlayweaver.sourceforge.net.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8356
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8356