# A Keyword Search Algorithm for Structured Peer-to-Peer Networks

Adriana Szekeres, Silviu Horia Baranga, Ciprian Dobre, Valentin Cristea
Faculty of Automatic Control and
Computer Science
University "Politehnica" of Bucharest
Romania, Bucharest
Emails: {adriana.szekeres, silviu.baranga}@gmail.com, {ciprian.dobre, valentin.cristea}@cs.pub.ro

*Abstract*—**Peer-to-Peer (P2P) networks are largely used for file-sharing and hence must provide efficient mechanisms for searching the files stored at various nodes. The existing structured P2P overlays support only "exact-match" look-up which is hardly sufficient in a filesharing network. This paper addresses the problem of keyword-based search in structured P2P networks. We propose a new keyword-based searching algorithm which can be implemented on top of any structured P2P overlay. We demonstrate that the proposed algorithm achieves very good searching results as it requires the minimum number of messages to be sent in order to find all the references to files containing at least the given set of keywords.**

*Keywords*—structured peer-to-peer networks; keyword search; dynamic distributed trees

## I. Introduction

Peer-to-peer (P2P) networks are amongst the most quickly growing technologies in computing. They are largely used for file-sharing, and hence must support a powerful mechanism for searching for the files stored at various nodes. In the last years, intense research has been done to overcome the scaling problems with unstructured P2P networks, such as Gnutella where data placement and overlay network construction are essentially random. A number of groups have proposed structured P2P designs. These proposals support a Distributed Hash Table (DHT) functionality. While there are significant implementation differences between these DHT-based networks, these systems all support (either directly or indirectly) a hash-table interface of $put(key, value)$ and $get(key)$. Moreover these systems are very scalable; look-ups can be resolved in $\log n$ (or $\log n^{\alpha}$ for small $\alpha$) overlay routing hops for an overlay network of size $n$ hosts. Thus, DHT-based networks largely solve the problem of scalability. However, DHTs support only "exact-match" look-ups, since they are hash-tables. This is fine for fetching files or solving domain names, but presents an even more impoverished query language than the original, unscalable P2P systems.

This paper addresses the problem of finding scalable keyword-based searching mechanisms in structured P2P networks. We propose a new keyword-based searching algorithm for structured P2P networks.

The remainder of the paper is organized as follows. In Section II we analyze the existing keyword-based algorithms. In Section III we present the new algorithm; we separately analyze the part of the algorithm which indexes the objects into the network and the part which searches for references to files described by a given set of keywords. We present the results in section IV, future work in section V and our conclusions in section VI.

## II. Previous Related Work

Most structured P2P overlay networks, [1], [2], [3], require that each node maintains a small routing table, while achieving $O(log_B N)$-messages lookup, where $N$ is the number of nodes in the network and $B$ varies from overlay to overlay (e.g. $N = 2$ in case of a Chord overlay, [1], and $N = 16$ in case of a Tapestry overlay with hexadecimal IDs, [3]). These structured P2P overlay networks basically support only exact name matching [4], [5], as objects are given a unique identifier obtained by hashing their names to determine their location in the network.

Most of the mechanisms proposed for keyword-based search in structured P2P networks use an inverted index as the primary data structure. An inverted index is a set of pairs $(w, O)$, where $w$ is a keyword, and $O$ is the set of objects containing this keyword. For example, given the objects: $s_0$ = "this is it", $s_1$ = "it was" and $s_2$ = "it is great", we can construct the following inverted file index (where the integers in the set notation brackets refer to the subscripts of the text symbols, $s_0$, $s_1$ etc.): 'this':{0}, 'is':{0,2}, 'it':{0,1,2}, 'was':{1}, 'great':{2}. To implement keyword search in a P2P network, a distributed version of inverted index can be built. The naive approach is to distribute the entries so that each keyword is assigned a node to index the objects that have this keyword. By incorporating into a P2P overlay network, a given keyword can be seen as key to determine the node that is responsible for the keyword, and obtain objects that contain the keyword. Using a $join$ operation, the objects containing a given set of keywords can be retrieved.

The keyword search method based on distributed index is commonly used in existing P2P systems (e.g., [6], [7], [8]). However, it has several drawbacks, as identified in [9] . The first problem concerns the load balancing. Few keywords occur very often while many others occur rarely so a node which is responsible for a very frequent keyword could be become over loaded. The second problem refers to the indexing mechanism. An object will be indexed at all the nodes responsible for each of its keyword. Although an object might be indexed at several nodes, the search queries which refer to that object are not distributed among the holding nodes. Also, a node might easily become a "hot spot". This means that if a node is responsible for a frequently searched keyword, like "mp3", it will become over-solicited.

All of the existing keyword search methods based on distributed index try to optimize the naive approach. In [6], a single entry in the inverted index corresponds to a set of maximum $m$ keywords, where m is a given parameter. So, when some query involving a keyword set $K$ is issued, instead of querying each individual keyword in $K$ and then taking a join operation, the set $K$ can be divided in disjunctive subsets of maximum $m$ keywords and query only the nodes responsible for these subsets. This saves bandwidth and speeds up search process, but at the expense of extra storage. In [7] the authors use bloom filters [10] to reduce bandwidth consumption. So, instead of transmitting the whole set documents which contain a certain keyword, needed for a $join$ operation, only the bloom filter associated to this set is transmitted. The peer node then computes the result of the $join$ operation. However, using the bloom filters imply a certain number of false positives. Bandwidth in query can also be reduced via caching [11].

Recent mechanisms for keyword-based search in structured P2P systems do not rely on distributed inverted index. In [9] the authors propose a novel indexing scheme which uses a hypercube to index objects according to their keyword sets. As in the methods based on distributed inverted index, an object can be retrieved after only one look-up on its set of keywords. However, when searching for the objects which contain at least the given set of keywords (superset search), multiple look-ups must be done. Basically, in this case, the search will be performed in the spanning tree constructed on the nodes pertaining to the sub-hypercube induced by the given set of keywords. Although the searching becomes more efficient when more keywords are given, the indexing scheme could be modified such that the spanning tree in which the searching is performed will contain only paths that are certain to lead to a reference of a previously indexed object. This will reduce the number of unnecessarily visited nodes when performing a superset search. Following this idea, we propose a new indexing scheme based on dynamically built trees, which will optimize the superset search.

## III. Proposed Algorithm

In this section we describe the keyword search algorithm. The proposed algorithm can be implemented on top of any structured peer-to-peer overlay (e.g Chord), which offers the mechanisms to route an overlay key to a network node. The algorithm uses dynamically built trees to optimize the number of messages sent during a keyword search.

### A. System definition

Let $N$ be the set of all nodes in the network, $S$ the set of all possible keywords, $P$ the set of all finite lexicographically ordered subsets of $S$, and $I$ the set of hash-keys.

We define the function $nodehash : N \mapsto I$, where $nodehash(x) = HV(x)$ (hash value of node $x$). As defined in the overlay layer, this hash function is injective.

In the same manner we define the function $sethash : P \mapsto I$, $sethash(x) = HV(x)$ (hash value of the lexicographically ordered set of keywords, $x$). This function may not be injective.

Let $lookup : I \mapsto N$, for which $lookup(K) = n$, where $n$ is the node responsible for the overlay key, $K$. This function returns the node which is responsible for a key and should be provided by any structured peer-to-peer overlay.

Let $node : P \mapsto N$, for which $node(s) = lookup(sethash(s))$. This function returns the node that is responsible for a set of keywords.

Let $prefix_n : P \mapsto P$, such that $prefix_n(\{k_1, \dots, k_m\}) = \{k_1, \dots, k_n\}$, where $k_1 < k_2 < \dots < k_m \land n \le m$ (by $<$ relation we refer to the lexicographical order).

Let $children : P \mapsto P$. The $children$ function and $P$ induce an oriented graph structure, where $P$ is the set of vertices. We denote this graph as the *search graph*. If $children(\{x_1, x_2, \dots, x_n\}) = \{q_1, q_2, \dots, q_m\}$, where $q_i \neq x_j$, $\forall i \leqslant n, \forall j \leqslant m$ , then $\forall X \in \{\{x_1, x_2, \dots, x_n\} \cup \{q_k\} | q_k \in \{q_1, q_2, \dots, q_m\}\}$ there exists an edge from $\{x_1, x_2, \dots, x_n\}$ to $X$.

**<u>Definition:</u>** A path in the *search graph* is an *ordered search path* if for every sequence $K_1 \mapsto K_2 \mapsto K_3$ in the path, such that $K_2 = K_1 \cup \{q_1\} \land K_3 = K_2 \cup \{q_2\}$, $q_1 < q_2$.

Let $parents : P \mapsto P$. The $parents$ function and $P$ also induce an oriented graph structure, which is the inverted graph induced by the $children$ function. We denote this graph as the *index graph*. If $parents(\{x_1, x_2, \dots, x_n\}) = \{q_1, q_2, \dots, q_m\}$ then $m \le n$, $\{q_1, q_2, \dots, q_m\} \subseteq \{x_1, x_2, \dots, x_n\}$ and $\forall X \in \{\{x_1, x_2, \dots, x_n\} \setminus \{q_k\} | q_k \in \{q_1, q_2, \dots, q_m\}\}$ there exists an edge from $\{x_1, x_2, \dots, x_n\}$ to $X$.

**<u>Definition:</u>** A path in the *index graph* is an *ordered index path* if for every sequence $K_1 \mapsto K_2 \mapsto K_3$ in the path, such that $K_1 = K_2 \cup \{q_1\} \land K_2 = K_3 \cup \{q_2\}$, $q_1 > q_2$.

Let $R$ be the set of all possible references.

Let $Q \subset P \times R$ be the set of all stored references in the system with their associated keyword set.

The system has the following properties (which will be demonstrated in the following sections):

- (Property 1) $\forall (s, l) \in Q$, there exists an ordered index path from $s$ to $K, \forall K \subset s$.
- (Property 2) $\forall K \in P, \forall (s, l) \in Q$ such that $K \subset s$, there exists an ordered search path from $K$ to $s$.

### B. Object Indexing

In this section we describe the operation of indexing an object reference, $(s, l)$ from $Q$. The algorithm builds and ad-hoc spanning tree in order to construct paths from object's set of keywords to all the subsets of the object's set of keywords. In this manner, the object can be retrieved by performing a *lookup* operation starting at each of these subsets. If a path for a keyword has already been constructed, it no longer needs to be. This results in lower indexing costs in a network for which there have already been indexed a large number of objects. Also, since keywords are usually related, there should be a correlation between the appearance of a keyword and the probability of appearance of another keyword, which would indicate that, in practice, the algorithm would often encounter cases where the path required for the indexing process has already been constructed by other object indexing operations. We describe the indexing algorithm in (Algorithm 1). The function $IndexObj$ is executed on node $n$ (the first parameter), and each call to this function is in fact a remote call to another node, which is done using the $lookup$ function provided by the overlay.

**Algorithm 1** IndexObj(node $n$, reference $r$, keyset $k$, key $absentKey$)

```
if $absentKey == NULL$ then
    $Q \leftarrow Q \cup (k, r)$
    for all key $a \in k$ do
        if $a \notin parents(k)$ then
            $parents(k) \leftarrow parents(k) \cup \{a\}$
            $IndexObj(node(k \setminus \{a\}), null, k \setminus \{a\}, a))$
        end if
    end for
else
    $children(k) = children(k) \cup \{absentKey\}$
    for all key $a \in k$ do
        if $a \notin parents(k)$ and $a < absentKey$ then
            $parents(k) \leftarrow parents(k) \cup \{a\}$
            $IndexObj(node(k \setminus \{a\}), null, k \setminus \{a\}, a))$
        end if
    end for
end if
```

(Fig 1.) shows the structure resulted after indexing an object with the associated keyword set $\{a, b, c\}$. First, a message is sent to $node(\{a, b, c\})$. Being the only node responsible for the subset of three of $\{a, b, c\}$, it will send messages to all its parents: $\{node(\{a, b\})$, $node(\{a, c\})$, $node(\{b, c\})\}$. When $node(\{a, b\})$ receives the message, it saves a reference to its child. Also, being the node responsible for the minimum subset of size two of $\{a, b, c\}$ (the *minimum parent* of $node\{a, b, c\}$) it sends a message to its parents: $\{node(\{a\}), node(\{b\})\}$. No further messages are sent from $\{node(\{a\})$ and $node(\{b\})\}$ as they don't have any possible parents. When $node(\{a, c\})$ receives the indexing message, it sends a message to its parent $node(\{c\})$, being the *minimum parent* of $node\{a, b, c\}$ which has $node(\{c\})$ as a parent (the *minimum child* of $node(\{c\})$). It does not send a message to $node(\{a\})$ because the minimum child of this node is $node(\{a, b\})$ which is the one responsible for sending a message to $node(\{c\})$.

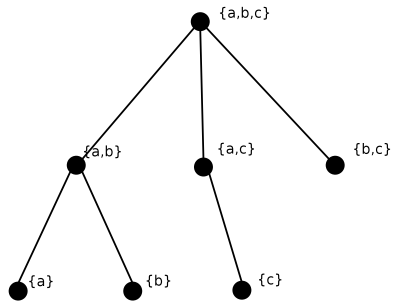

Fig. 1. Indexing of $\{a, b, c\}$

In (Fig. 2) it is shown the indexing process of an object with the associated keyword set $\{a, b, c, d\}$. We assume that an object with the associated keyword set $\{a, b, c\}$ has already been indexed. First, a message is sent to $node(\{a, b, c, d\})$. After receiving the message, it sends a message to all its parents: $\{node(\{a, b, c\})$, $node(\{a, b, d\})$, $node(\{a, c, d\})$, $node(\{b, c, d\})\}$. When $node(\{a, b, c\})$ receives the message, it saves a reference to this child and verifies if it has any entries for the same set of keywords ($\{a, b, c\}$). Because another object has been indexed for this set of keywords, it does not send a message to its parents. The rest of the indexing process is done as described previously for (Fig.1). Notice that the number of messages needed to index an object with the associated set of keywords, $K$, will decrease if there have already been indexed objects with the associated set of keywords, $K_O$, such that $K_O \subseteq K$ or $K \subseteq K_O$ .

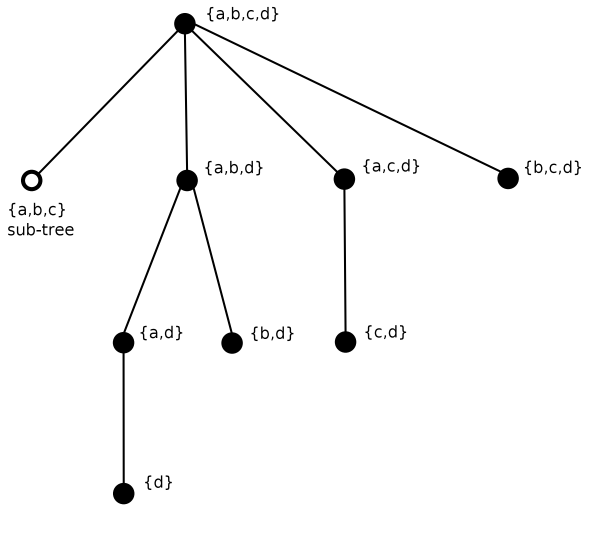

Fig. 2. Indexing of $\{a, b, c, d\}$

**Lema 1.** $\{\{k_1, \ldots, k_n\}\} \cup \bigcup_{h=1}^{n} \{\{X \cup Y\} | X \in \mathcal{P}(\{k_1, \ldots, k_{h-1}\}) \wedge Y = \{k_{h+1}, \ldots, k_n\} \wedge X \cup Y \neq \emptyset\} = \mathcal{P}(\{k_1, \ldots, k_n\}) \setminus \{\emptyset\}$, where $\mathcal{P}(\{x_1, \ldots, x_n\})$ represents the set of all subsets of $\{x_1, \ldots, x_n\}$, including $\emptyset$.

**Proof:** (Proof by using the Double Inclusion Theorem) Let $A = \{\{k_1,\ldots,k_n\}\} \cup \bigcup_{h=1}^{n}\{\{X \cup Y\}|X \in \mathcal{P}(\{k_1,\ldots,k_{h-1}\}) \wedge Y = \{k_{h+1},\ldots,k_n\}\}$ and $B = \mathcal{P}(\{k_1,\ldots,k_n\})$.

1. Prove that "$A \subseteq B$": This is obviously true as $A$ contains only sets formed using elements from $\{k_1,\ldots,k_n\}$.

2. Prove that "$B \subseteq A$": Let $T = \{k_{i_1},\ldots,k_{i_m}\}$, $T \in B \setminus \{\{k_1,\ldots,k_n\}\}$. Let $p = max\{i|1 \leqslant i \leqslant n \wedge k_i \notin T\}$. As $|T| < n$, $p$ exists $\Rightarrow \{k_{p+1},\ldots,k_n\} \subset T$. $T \setminus \{k_{p+1},\ldots,k_n\} \in \mathcal{P}(\{k_1,\ldots,k_{p-1}\}) \Leftrightarrow T \in \{\{X \cup Y\}|X \in \mathcal{P}(\{k_1,\ldots,k_{p-1}\} \wedge Y = \{k_1,\ldots,k_{p-1}\}\} \Rightarrow T \in \bigcup_{h=1}^{n}\{\{X \cup Y\}|X \in \mathcal{P}(\{k_1,\ldots,k_{h-1}\}) \wedge Y = \{k_{h+1},\ldots,k_n\}\} = A \setminus \{\{k_1,\ldots,k_n\}\}$ (obviously, if $T = \{k_1,\ldots,k_n\}$ then $T \in A$) $\Rightarrow \forall T \in B, T \in A$. **q.e.d.**

**Lemma 2.** Let $K = \{k_1,\ldots,k_n\}$ be a set of keywords of size $n$. If $k_x \in parents(K)$ (there exists an edge from $K$ to $K \setminus \{k_x\}$) then there exist ordered index paths from $K$ to each $z \in \{\{X \cup Y\}|X \in \mathcal{P}(\{k_1,\ldots,k_{x-1}\}) \wedge Y = \{k_{x+1},\ldots,k_n\} \wedge X \cup Y \neq \emptyset\}$.

**Proof:** (Proof by Mathematical Induction) Let P(n) be the statement "Lemma 2 is true for all sets of keys of size n".

**Initial Step.** P(1) is true because $parents(\{k_1\}) = \emptyset$. We show that P(2) is also true. Let $K = \{k_1, k_2\}$. If $k_1 \in parents(K)$ then there is an edge to $\{k_2\}$. According to (Lemma 2) there must be ordered index paths form $K$ to each $z \in \{\{X \cup Y\}|X \in \mathcal{P}(\emptyset) \wedge Y = \{k_2\} \wedge X \cup Y \neq \emptyset\} = \{k_2\}$, which is true. If $k_2 \in parents(K)$ then there is an edge to $\{k_1\}$. According to (Lemma 2) there must be ordered index paths from $K$ to each $z \in \{\{X \cup Y\}|X \in \mathcal{P}(\{k_1\}) \wedge Y = \emptyset \wedge X \cup Y \neq \emptyset\} = \{k_1\}$, which is true.

**Inductive Step.** We assume that P($i-1$) holds and we prove that P($i$) also holds. Let $K = \{k_1,\ldots,k_i\}$. Let $k_x \in parents(K)$. As $k_x \in parents(K)$ then, according to the algorithm, $\{k_1,\ldots,k_{x-1}\} \subseteq parents(K \setminus \{k_x\})$. In other words, in function $IndexObj$ executed at $node(K \setminus \{k_x\})$, when the $absentKey = k_x$, parents $k_1,\ldots,k_{x-1}$ will be added (if they have not been previously added) as they are smaller than $k_x$. This function is indeed executed because, in the algorithm, adding a parent is immediately followed by a call to $IndexObj$ at that parent; in this particular case, after adding $k_x$ as a parent at $parents(K)$ the call to this function follows. Also, edges to this child $(K \setminus \{k_x\})$ will be added as it calls $IndexObj$ for each of its parents and they will execute the instruction $children(k) = children(k) \cup \{absentKey\}$.

So, in other words, there exists an edge from $K \setminus \{k_x\}$ to each $z \in \bigcup_{h=1}^{x-1}\{K \setminus \{k_x, k_h\}\}$ (its parents). Because $|K \setminus \{k_x\}| = i - 1$ and P($i-1$) holds then, as there exist edges to each parent $k_h$, then there exists ordered index paths from $K \setminus \{k_x\}$ to each $z \in Z = \bigcup_{h=1}^{x-1}\{\{X \cup Y\}|X \in \mathcal{P}(\{k_1,\ldots,k_{h-1}\}) \wedge Y = \{k_{h+1},\ldots,k_{x-1},k_{x+1},\ldots,k_i\} \wedge X \cup Y \neq \emptyset\}$. To prove that P($i$) also holds we need to show that $Z \cup \{K \setminus \{k_x\}\}$ (we must also add the edge to $K \setminus \{k_x\}) = \{\{X \cup Y\}|X \in \mathcal{P}(\{k_1,\ldots,k_{x-1}\}) \wedge Y = \{k_{x+1},\ldots,k_i\} \wedge X \cup Y \neq \emptyset\}$. $Z$ can also be written: $Z = \bigcup_{h=1}^{x-1}\{\{X \cup W \cup Y\}|X \in \mathcal{P}(\{k_1,\ldots,k_{h-1}\}) \wedge W = \{k_{h+1},\ldots,k_{x-1}\} \wedge Y = \{k_{x+1},\ldots,k_i\} \wedge X \cup W \cup Y \neq \emptyset\}$, so it is sufficient to show that $\{\{k_1,\ldots,k_{x-1}\}\} \cup \bigcup_{h=1}^{x-1}\{\{X \cup Y\}|X \in \mathcal{P}(\{k_1,\ldots,k_{h-1}\}) \wedge Y = \{k_{h+1},\ldots,k_{x-1}\} \wedge X \cup Y \neq \emptyset\} = \mathcal{P}(\{k_1,\ldots,k_{x-1}\}) - \{\emptyset\}$. This is true, according to (Lemma 1).The paths are obviously ordered index paths because $k_x > k_h, \forall k_h \in parents(K \setminus \{k_x\})$ (the sequence $K \mapsto K \setminus \{k_x\} \mapsto K \setminus \{k_x, k_h\}$ also respects the condition from the definition of *ordered index path*). So P($i$) also holds. **q.e.d.**

**Theorem 1.** A system that holds (Property 1) and (Property 2) will also hold (Propriety 1) and (Property 2) after the indexing operation.

**Proof:** To prove that (Property 1) is preserved after an indexing operation, we must notice that when indexing a reference, which has associated the keywords $K = \{k_1,\ldots,k_n\}$, function $IndexObj$ will be called with $absentKey = NULL$ and executed on $node(K)$, thus adding all the possible parents of $K$. So, $parents(K)$ will be $\{k_1,\ldots,k_n\}$. According to (Lemma 2), as $\forall h$ such that $1 \leqslant h \leqslant n, k_h \in parents(K)$, then there exist ordered index paths from $K$ to each $z \in Z = \bigcup_{h=1}^{n}\{\{X \cup Y\}|X \in \mathcal{P}(\{k_1,\ldots,k_{h-1}\}) \wedge Y = \{k_{h+1},\ldots,k_n\} \wedge X \cup Y \neq \emptyset\}$. According to (Lemma 1), $Z \cup \{K\} = \mathcal{P}(K) \setminus \{\emptyset\}$. This means that there exist paths from $K$ to all subsets of $K$, thus (Property 1) is preserved after an indexing operation.

To prove that (Property 2) is also preserved after an indexing operation, we must first note that for each edge in the *index graph* there exists a corresponding inversed edge in the *search graph*. This is a trivial consequence of the algorithm, since each element added to the parent set, will have a corresponding added element in the children set in the next call to the $IndexObj$ method, executed on the node responsible for the parent. Since (Property 1) holds, (Property 2) also holds, since there exists an ordered search path from $K$ to $s$, which is exactly the inverse of the existing ordered index path from $s$ to $K$. **q.e.d.**

Note that the number of messages sent during an indexing operation is equal to the number of the subsets of the set of keywords associated to the reference that is being indexed, if there were no previous indexing operations. The height of the tree gives us the temporal complexity of this algorithm, which is $O(|\ |k\ |\ |)$, where $k$ is the set of keys for which we insert the reference. We also note that we store the reference only at the root of the tree. This allows us to save space used for reference storage. The number of nodes at which the reference is inserted depends on the overlay used. In Chord, we the key is inserted at a single location. In Kademlia, the key is stored at 10 locations, for the default Kademlia settings. In case of using Kademlia as the overlay, fault tolerance will be automatically introduced as a property of the keyword search network.

## *C. Search Method*

In this section we describe the process of keyword-based searching in the peer-to-peer network. An object $O$, with its associated set of keywords $K_O$, is said to be described by a set of keywords $K$, if $K \subseteq K_O$.

Given a set of keywords $K$, the objective of the search algorithm is to find all the objects described by the given set of keywords.

In figure (Fig. 3) we show how the search algorithm works. We assume that three objects, with the associated set of keywords $\{a, b, c, d\}$, $\{d, e, f\}$ and $\{d, e\}$, have already been indexed. The objective is to find all the references to objects described by $\{d\}$. Since $node(\{d\})$ is the only node receiving the message, it will send a message to all its children: $node(\{a, d\})$, $node(\{d, e\})$ and $node(\{d, f\})$. $node(\{a, d\})$ will send a message to $node(\{a, b, d\})$ since it is its only child ( and it would have also been the *minimum child* since 'd' is lexicographically greater then both 'a' and 'b'). In the same manner, $node(\{a, b, d\})$ will send a message to $node(\{a, b, c, d\})$ and $node(\{d, e\})$ will send a message to $node(\{d, e, f\})$. A special case is happening at $node(\{d, f\})$. Although $node(\{d, e, f\})$ is a child of $node(\{d, f\})$, a message from $node(\{d, f\})$ to $node(\{d, e, f\})$ will not be sent, because using the initial set, the node decides that there already exists a node that is lexicographically smaller, $node(\{d, e\})$, that will send the message to $node(\{d, e, f\})$. Since it does not have any more children to send messages to, $node(\{d, f\})$ will send a reply to the search call containing the object stored locally. All nodes merge the response messages and the requester receives the response with all the stored values in the network which contain keywords $\{d\}$.

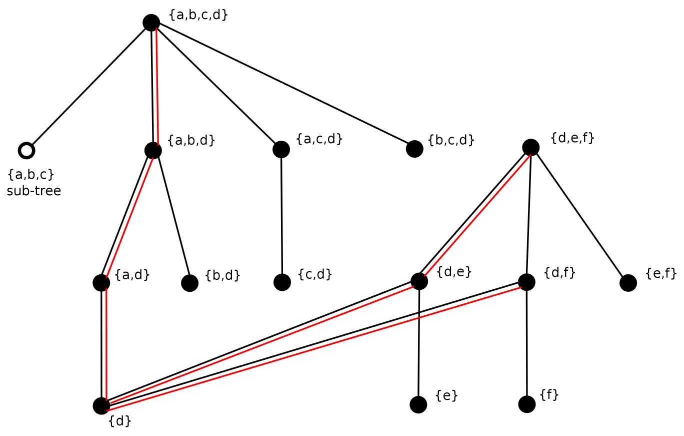

Fig. 3. Searching process for objects described by $\{d\}$

The algorithm is described in (Algorithm 2). The parameters for this search algorithm are a node $n$, a key-set $k$, and a key-set $initialSet$. Each call to this function must must hold the assertion $n = node(k)$. This means that $Search$ function is executed on node $n$, which is responsible for the key-set $k$ and a call to this function is in fact a remote call, implemented using the *lookup* mechanism provided by the overlay.

**Algorithm 2** referenceSet $Search$(node $n$, keyset $k$, keyset $initialKeyset$)

```
referenceSet $result \leftarrow \{i \in R | (k, i) \in Q\}$
for all key $a \in children(k)$ do
    bool $follow \leftarrow true$
    for all key $b \in k \setminus initialKeyset$ do
        if $a < b$ then
            $follow \leftarrow false$
        end if
    end for
    if $follow == true$ then
        $result \leftarrow result \cup Search(node(k \cup \{a\}), k \cup \{a\}, initialSet)$
    end if
end for
return $result$
```

**Lemma 3.** Let $K$ be a set of keywords situated on a path followed by the search algorithm applied for the $initialKeyset = I$. The following properties hold:

I. If $a \in children(K)$ then $\exists(s, l) \in Q$ such that $prefix_{||K \setminus I||+1}(s \setminus I) = K \cup \{a\} \setminus I$.

II. Let $(s, l) \in Q$ such that there exist two paths, $x_1 \mapsto \ldots \mapsto x_n$ and $x_1' \mapsto \ldots \mapsto x_n'$, for which $x_1 = x_1' = K \wedge x_n = x_n' = s$, followed by a search operation (according to the $Search$ algorithm an edge is followed only if the corresponding child, $a$, is bigger (lexicographically) then any $b \in k \setminus initialKeyset$ ). Then $x_i = x_i', \forall 1 \leq i \leq n$.

III. The search algorithm finds all the references $(s, l) \in Q$ such that $I \subset s$.

**Proof:**

I. If $I$ is the $initialKeyset$ received at $node(K)$ during a search operation then, according to the $Search$ algorithm, there exists a path from $I$ to $K$: $I \mapsto I \cup \{q_1\} \mapsto \ldots \mapsto K = I \cup \bigcup_{i=1}^{n} \{q_i\}$, where $q_1 < \ldots < q_n$ (lexicographically ordered). If $a \in children(K)$ then, as the edge was added by an index operation, two of the following situations can be possible:

- An indexing operation was initiated ($absetKey = NULL$) at $node(K \cup \{a\}) \Rightarrow \exists(s, l) \in Q$ such that $s = K \cup \{a\}$. Because $prefix_{|K \setminus I|+1}(s \setminus I) = prefix_{|K \setminus I|+1}(K \cup \{a\} \setminus I) = K \cup \{a\} \setminus I$, the property holds.
- An indexing operation was executed at $node(K \cup \{a\})$ for $absentKey = h_1$, such that $h_1 > a$. Recursively, because $h_1 \in children(K \cup \{a\})$, again these both situations could be possible $\Rightarrow \exists(s, l) \in Q$ such that $s = K \cup \{a\} \cup \bigcup_{i=1}^{m} \{h_i\} \wedge h_1 < \ldots < h_m$. Because $prefix_{|K \setminus I|+1}(s \setminus I) = prefix_{|K \setminus I|+1}(K \cup \{a\} \bigcup_{i=1}^{m} \{h_i\} \setminus$

$I) = K \cup \{a\} \setminus I$, as $q_i < a < h_j, \forall i \leq n, j \leq m$, the property holds.

II. Let $x_{i+1} - x_i = \{a_i\}, x^{'}_{i+1} - x^{'}_i = \{a^{'}_i\}, \forall 1 \leq i \leq n-1$. According to the search algorithm, $a_i < a_{i+1}, a^{'}_i < a^{'}_{i+1} \forall 1 \leq i \leq n-1$. Since $\bigcup_{i=1}^{n}\{a_i\} = \bigcup_{i=1}^{n}\{a^{'}_i\} = s \setminus K$ then $a_i = a^{'}_i, \forall 1 \leq i \leq n-1 \Rightarrow x_i = x^{'}_i, \forall 1 \leq i \leq n$.

III. Because the search algorithm follows all the ordered search paths form $I$ (initially, it follows the edges to all its children; then, for each child it follows the edges to those children bigger than all the previous followed children and so on) then, from (Property 2) it results that all the nodes containing $(s, l)$, for which $I \subset s$, will be visited. **q.e.d.**

**Theorem 2.** The search algorithm applied for $initialKeyset = I$ finds all the references $(s, l) \in Q$ such that $I \subset s$, by propagating a search message in a tree, rooted at $I$, in which for every leaf, $L$, there exists $(s, l) \in Q$ such that $L = s$.

**Proof:** This is a consequence of (Lemma 3) which states that: (II+III) the search algorithm follows unique paths from $I$ to every $(s, l) \in Q$ for which $I \subset s$; (I) a path which doesn't lead to a reference $(s, l) \in Q$, for which $I \subset s$, is not followed. **q.e.d.**

Since the messages are sent in a spanning tree, in which each leaf contains at least one entry, we optimize the number of search messages that are sent through the system. There are no unnecessary messages sent during a get request. Each response comes from a single source. The response may contain identical references if the object was indexed multiple times with different keywords. If a search message is forwarded from one node, it is done so with the certainty that following this path, at least one result for the keyword search can be reached.

## IV. Results

We have implemented and simulated the proposed peer-to-peer keyword search algorithm using the Oversim framework [12]. As the overlay network, we used the existing Chord implementation for Oversim, which offers the $look - up$ operation. For the hashing function we used the SHA-1 algorithm, and we hashed the string composed of all of the sorted keywords. As the data set, we represented the keywords as integers for better simulation capabilities. The data we used was randomly generated and the keywords were not correlated. The use of a correlated set of keywords generator would have produces better results.

Note that since our indexing scheme is independent of the underlying DHT overlay, our experiments were conducted only on the the logical structure we built. Thus, all measurements are with respect to the structure constructed on top of the overlay.

The first experiment we conducted confirmed a property of the indexing algorithm: the number of messages sent in order to solve a put request for a set of keywords $K$, decreases if there are already indexes to subsets or supersets of $K$. The simulation consisted of 100 nodes. Each node periodically sent a put request for a subset of a predefined set of 10 keywords ($\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$). This way, the edges between the parents and children were quickly formed, thus decreasing the overhead for subsequent requests. (Fig. 4) illustrates the number of put messages generated, measured on a 5 to 5 seconds basis. It can be seen that the number of put messages sent decreases. In (Fig. 5) is shown the average delay to solve a put request, also recorded from 5 to 5 seconds.

For the second experiment we built a scenario consisting of 1000 nodes. We used the GNU Scientific Library for the test generation. The exponential distribution for the frequency of the words had an exponential factor of 1.5, a scalar factor of 1000 and a maximum size limit of 5000 possible keywords. The size of the keyword set was also randomly generated and had a size between 5 and 10, with a gaussian distribution over this interval. The data for the searching algorithm was taken from previous put operations, such that every search has at leas one result returned. The size of the searching set was always smaller than the size of the indexing set, and was chosen randomly. We observe that for a smaller searching set, the searching operation becomes more complex, since it involves more search calls on different nodes. The simulation run for approximately 1000 simulated seconds. For the first 400 simulated seconds, all the nodes periodically generated put requests. For the rest of the time, the nodes started to periodically generate get requests. This allowed us to evaluate the performance of the indexing algorithm and the performance of the searching algorithm separately. In (Fig. 6) is illustrated the number of generated put messages. It can be noticed that the average number of messages sent in the network was approximately 2400 messages per second. This is an acceptable value considering the size of the network. In (Fig. 7) is illustrated the average Put Request Delay (in seconds), for the simulated network. As the graphic shows, put request takes, in average, almost 2 seconds. The maximum delay to solve a put request is almost 3 seconds, as shown in (Fig. 8). The put request execution time is greatly influenced by the state of the network and the context in which it is done. If the network had already made the connections required for this operation, the operation time will be low. Else, we encounter a large deviation from the mean when encountering requests that need to explore a keyspace where there have not been previously made an average set of requests.

In (Fig. 9) is shown the average number of get request messages, measured from 5 to 5 seconds, for the simulated network of 1000 nodes. It can be noticed that the average number of get messages sent was 500 get messages per second. This clearly shows that a search operation is very efficient, much more efficient than a put operation. This is also showed in (Fig. 10) which illustrates the average delay of a get request: 1.2 seconds. The maximum delay for solving a get request is approximately 3 seconds.

## V. Future Work

As we have mentioned in the *Object Indexing* section, the performance of the application is dependent to the overlay net-

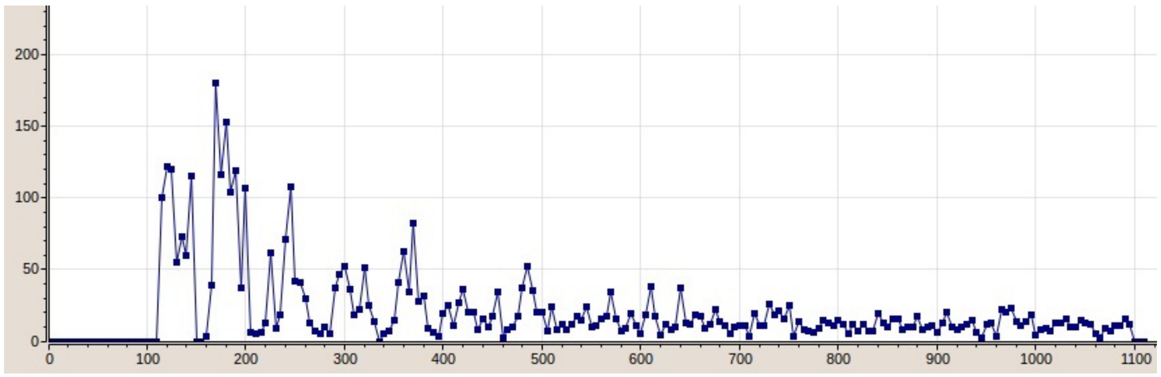

Fig. 4. Number of put request messages in time (seconds) - measured from 5 to 5 seconds - for a network with 100 simulated nodes and 10 possible keywords

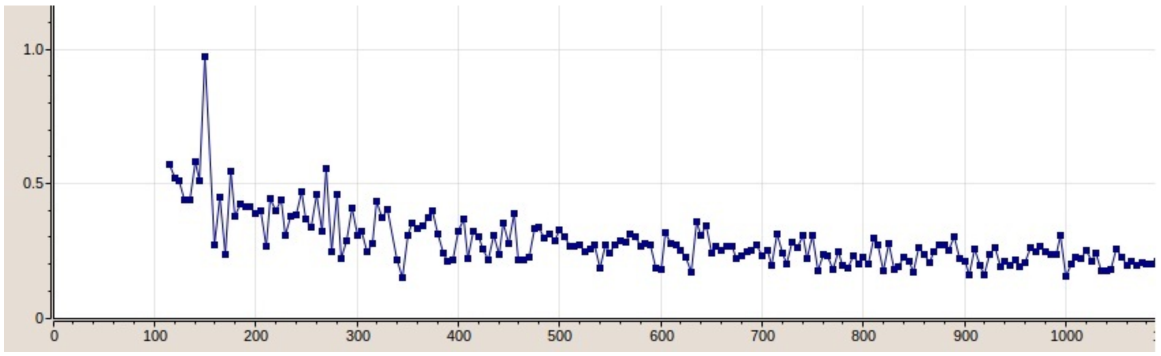

Fig. 5. The average Put Delay (seconds) for a network with 100 simulated nodes and 10 possible keywords

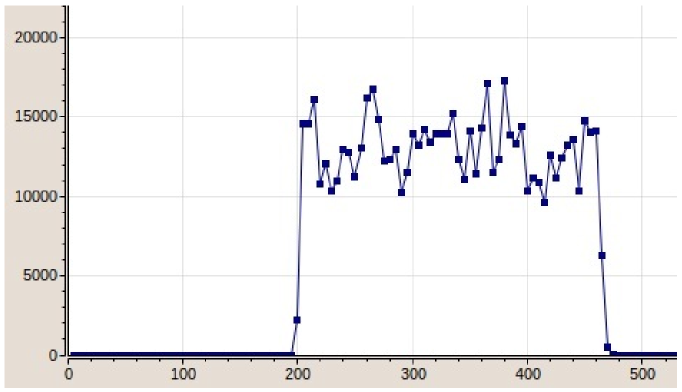

Fig. 6. Number of put request messages in time (seconds) - measured from 5 to 5 seconds - for a network with 1000 simulated nodes and 5000 possible keywords

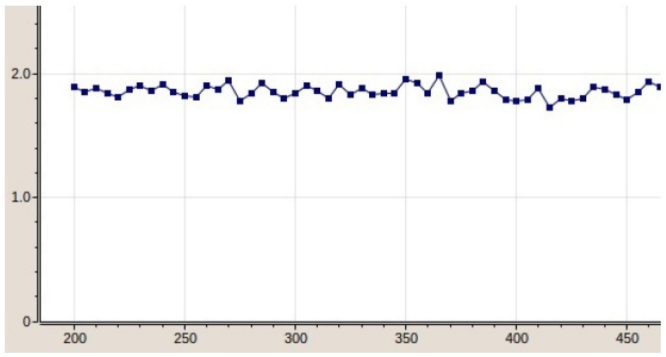

Fig. 7. The average Put Delay (seconds) for a network with 1000 simulated nodes and 5000 possible keywords

work used. Since we have tested the application using Chord, we would like to also test it using other DHT implementations. Secondly we think that the results would be better using data traces from real applications as the input for the simulator, since this data would exhibit word associations. The most important improvement to our current algorithm would be the reduction of messages for both put and get operations. We think that this could be done naturally using a Kademlia overlay instead of Chord and a locality preserving hashing function, since this would allow close data sets to be clustered into nodes, and thus bring a great reduction of the number of messages in both indexing and searching operations.

## VI. Conclusion

We have proposed a peer-to-peer keyword-based search algorithm, which can be used in conjunction with any structured peer-to-peer overlay networks. The algorithm is scalable to the number of peers in the system, since the temporal complexity of both the indexing and searching operations do not depend on the number of peers in the system. We have shown that the searching operation is very efficient, having the property of

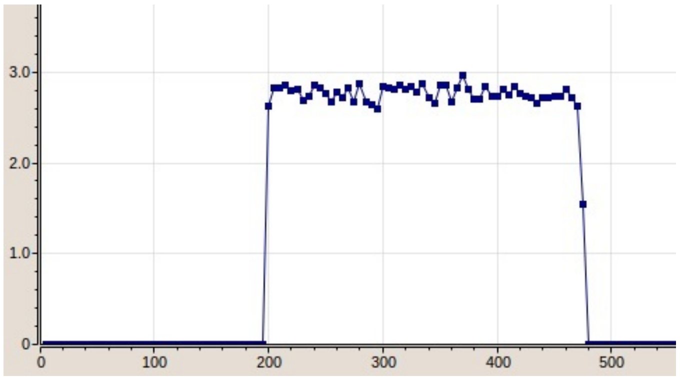

Fig. 8. The maximum Put Delay (seconds) for a network with 1000 simulated nodes and 5000 possible keywords

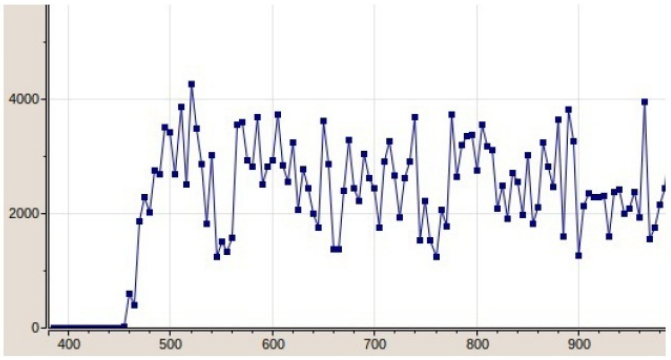

Fig. 9. Number of get request messages in time (seconds) - measured from 5 to 5 seconds - for a network with 1000 simulated nodes and 5000 possible keywords

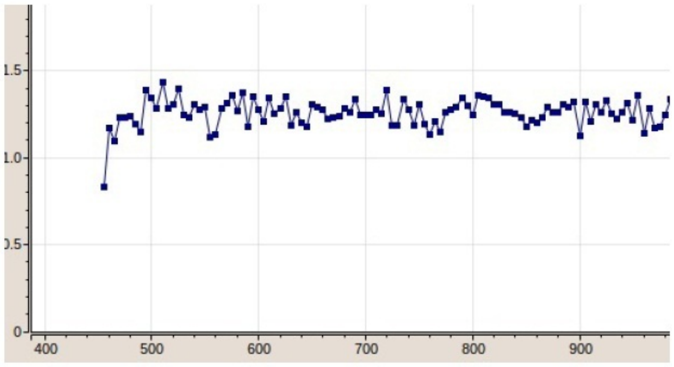

Fig. 10. The average Get Delay (seconds) for a network with 1000 simulated nodes and 5000 possible keywords

sending a number of messages in the system that depend on the size and content of the result set. Also, the cost of indexing operation is acceptable, decreasing as the system holds more information.

## VII. Acknowledgements

This work was supported by CNCSIS-UEFISCSU, project "TRANSYS Models and Techniques for Traffic Optimizing in Urban Environments", number PN II-RU code/2010, "CNCSIS-PD" ID: 238.

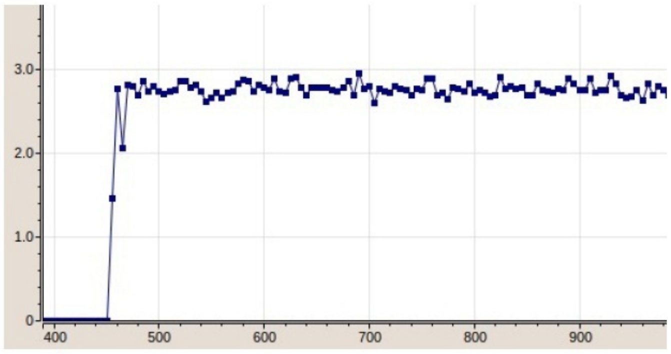

Fig. 11. The maximum Get Delay (seconds) for a network with 1000 simulated nodes and 5000 possible keywords

## References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2001, pp. 149–160.

[2] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. London, UK: Springer-Verlag, 2002, pp. 53–65.

[3] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and," Berkeley, CA, USA, Tech. Rep., 2001.

[4] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica, "Complex queries in dht-based peer-to-peer networks," in *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. London, UK: Springer-Verlag, 2002, pp. 242–259.

[5] J. Li, B. T. Loo, L. Joseph, J. M. Hellerstein, D. R. Karger, R. Morris, and M. F. Kaashoek, "On the feasibility of peer-to-peer web indexing and search," in *In IPTPS03*, 2003, pp. 207–215.

[6] O. D. Gnawali, "A keyword-set search system for peer-to-peer networks," Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, United States, 2002.

[7] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," in *Middleware '03: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 21–40.

[8] S. Shi, G. Yang, D. Wang, J. Yu, S. Qu, and M. Chen, "Making peer-to-peer keyword searching feasible using multi-level partitioning," in *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS 2004)*. Springer-Verlag New York, Inc., 2004, pp. 151–161.

[9] Y.-J. Joung, C.-T. Fang, and L.-W. Yang, "Keyword search in dht-based peer-to-peer networks," in *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 339–348.

[10] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[11] B. Bhattacharjee, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi, "Efficient peer-to-peer searches using result-caching," in *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, February 2003.

[12] I. Baumgart, B. Heep, and S. Krause, "OverSim: A scalable and flexible overlay framework for simulation and real network applications," in *Ninth International Conference on Peer-to-Peer Computing (IEEE P2P*, Sep. 2009, pp. 87–88.