Available online at www.sciencedirect.com
ScienceDirect
Computer Communications 31 (2008) 423–436
computer communications
www.elsevier.com/locate/comcom

# Scalable tag search in social network applications

Alberto Mozo [a,*], Joaquín Salvachúa [b]

[a] Department of Arquitectura y Tecnología de Computadores, Universidad Politécnica de Madrid (UPM), Madrid, Spain
[b] Departament of Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid (UPM), Madrid, Spain

Available online 2 September 2007

**Abstract**

Emerging social network applications for sharing and collaboration need a way to publish and search a big amount of social objects. Currently, social applications allow the association of a set of user defined keywords, named tags, when publishing these objects, in order to allow searching for them later using a subset of these tags. Commercial systems and recent research community proposals preclude a wide Internet deployment due to the emergence of scalability and hot spot problems in the nodes.

We propose T-DHT, an innovative hybrid unstructured–structured DHT based approach, to cope with these high demanding requirements, in a fully scalable, distributed and balanced way. The storage process allows attaching a set of user tags to the stored object and takes at most O(Log($N$)) node hops. The tag information attached to the object is stored in a compact way into the node links using a bloom filter, in order to be used later in the search process. The search process allows searching for previously stored objects by means of a tag conjunction and also takes at most O(Log($N$)) node hops. The search process is based on DHT typical search combined with an unstructured search algorithm using the tag information previously stored into bloom filters of node links. The simulation results show T-DHT performs in a fully balanced and scalable way, without generating typical hot spot problems even if unbalanced distributions of popular tags are used.

Although T-DHT has been devised to build a scalable infrastructure for social applications, it can be applied to solve the more general Peer-to-Peer keyword search problems.
© 2007 Elsevier B.V. All rights reserved.

*Keywords:* Distributed algorithms; Communication protocols; Peer-to-Peer; Very large database systems; Keyword search

## 1. Introduction

New applications and services are now conquering the audience attraction in Internet: the so-called social applications. The change of focus from the system to the people is something that allows people an easier way to collaborate in the web. Flickr, the social application for store, search, sort and share photos; Del.icio.us, the famous social bookmarking application; and YouTube the popular video sharing website where users can upload, view and share video clips, are paradigms of this new type of applications. In these applications, one of the methods to help people to classify and locate information in this so-called Web 2.0 paradigm is the use of folksonomies [1]. Here, instead of using static formal defined ontologies, the people associate a set of variable and user defined tags to different contents. The social applications usually need a way to publish and locate a big amount of so-called social objects and their attached tags. Nowadays, we must take into account that this Web 2.0 paradigm does not involve too many technical changes, because it is just a new way of using previously existent technologies, and so the storage for such contents is usually implemented using a relational database that is not tailored to this kind of usage, and so, it does not scale adequately. If we want to build a new infrastructure layer for these new applications we must look them from a completely different approach, and so, we should be able to support the exponential growth of user generated contents (like blogs, RSS sources and podcast contents).

* Corresponding author. Tel.: +34 91 336 5118.
*E-mail addresses:* amozo@eui.upm.es (A. Mozo), jsalvachua@dit.upm.es (J. Salvachúa).

0140-3664/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2007.08.035

Currently, commercial solutions relay in database centric approaches, and they are dying of success because of their limited scalability. Trying to solve this problem, distributed database systems have been a hot topic of interest in the database research community during the last years. Some approaches have focused on how to make transparent the distribution of data behind a more or less SQL standard query language, sacrificing scalability. SDD-1 [3] and Mariposa [2] are examples of it, scaling at most a few hundreds of nodes.

Recent research community proposals are focusing on Peer-to-Peer storage infrastructures in order to provide a distributed, balanced and high scalable publish and search solution. Distributed hash tables (DHT) Chord [4], CAN [5], Pastry [6] and Tapestry [7], among others, provide a pretty good method to store and search data in a distributed, balanced and scalable way. However, the keyword search is difficult to implement due to the fact the search procedure needs to know the object key identifier (usually obtained using SHA-1 hash function on the object). This key identifier is needed in order to route adequately the search towards the node storing the object. However, in social networks the search application only has available a set of user defined tags describing the object, and does not have a direct way to obtain the object key identifier.

Several solutions have been devised ([8,25–27] among others) based on distributed inverted index. Besides publishing the object, a mapping between each keyword identifier and the object identifier is also published. Subsequently, the search procedure is achieved in two steps: (a) we locate and receive the object identifiers associated to each target tags; (b) we get the objects fulfilling the tag conjunction, joining the results obtained in the former step. This approach solves partially the problem but incurs in two inefficiencies. First, the popular tags defined by users will generate hot spots in the nodes responsible for storing the popular tags. These nodes will be responsible for storing and answering to a huge amount of search requests. Also the links and their neighbor nodes will suffer the unbalanced number of store and search requests directed to the former nodes. And second, the post-join process needed to get the subset of objects fulfilling the tag conjunction, forces to transmit into the network a huge amount of results that will be discarded subsequently, because they do not fulfill the tag conjunction, hence wasting bandwidth network resources. Some query-cache procedures are proposed [9] in order to mitigate the hot spot problem. Also [26] proposes additional publishing of two or three keyword combinations, instead of a single keyword, diminishing the number of results received when requesting for popular tags. Nevertheless, the scalability problem remains and precludes a wide Internet deployment.

We propose a new approach to achieve real scalability in the store and search procedure. We have developed an innovative technique modifying a typical DHT overlay in order to maintain the high quality properties when storing objects, but allowing a conjunctive tag driven search while maintaining the scalability properties of DHTs.

The storage process of an object allows attaching a set of user tags to the stored object. The storage process is based on typical DHT procedures, but slightly modified in order to also insert the object tag information. During the storage process, a publishing sub-phase is started where the object attached tag information is inserted into the node links, in order to adequately drive the later search procedure. Since the store and publish procedures are DHT based, the whole storage process takes at most $O(\mathrm{Log}(N))$ node hops.

The search process allows finding previously stored objects by means of tag conjunction. This process is based on typical DHT search combined with an unstructured locate algorithm using the tag information previously stored into the bloom filter of node links. Therefore, when a search procedure is started, the tag information previously inserted into the node links is consulted in order to determine the right path to reach the object that fulfills the tag conjunction. In this way, the locate procedure behaves similarly to an unstructured Peer-to-Peer overlay search procedure (e.g. Gnutella [10] without the scalability problems associated to this kind of systems [11]). Additionally, since the search and locate procedures are DHT based, the whole search process is bounded to $O(\mathrm{Log}(N))$ node hops.

In order to insert in a compact way and recover efficiently routing tag information in the links, we associate a bloom filter [12] to each node link. Since the bloom filter size is predetermined and constant at startup phase, we guarantee the system scalability, with a little tradeoff due to the probability of false positive generation, during the search routing process. These false positives are eliminated with a high probability in subsequent overlay hops and do not degrade substantially the system performance.

In T-DHT there is no need to generate expensive inverted indices in order to be able to search for tag conjunctions. Besides, no post-join operations are required in the source node to achieve the object subset fulfilling the tag conjunction constraints. The simulation results have shown that T-DHT performs storing and searching in a fully balanced, distributed and scalable way, avoiding the hot spots in nodes. Even if the tag distribution is composed by popular tags following an unbalanced Zipf distribution, the performance of conjunctive tag search achieves identical results to those obtained using a well balanced random tag distribution.

In the next section we introduce to the basis of T-DHT architecture and protocol. We describe the detailed T-DHT routing algorithm in the store and publish phase and subsequently the search and locate procedures. In Section 3, some protocol extensions are presented. Section 4 presents the results of the first working prototype and Section 5 shows the related works. Finally we conclude in Section 6.

## 2. T-DHT architecture

T-DHT belongs to the family of Peer-to-Peer content distribution networks (CDN). It is formed by a set of collaborative peer nodes composing an application overlay. T-DHT allows the storage and search of objects (or references to them) into the overlay network. Associated with the objects, the user defines and attaches a set of keywords named tags. There is not a common ontology to describe the valid tags but they are character strings with free syntax and semantics. A natural and adaptive process inside the social network will make more accessible the objects with popular and well defined tags, and also this adaptive process will hide the objects with bad and obscure defined tags. A user can search object references inside the T-DHT network using a conjunctive subset of tags that the target objects must fulfill.

In the next subsection we will describe the store and publish protocol, and subsequently the search and locate procedure will be shown.

### 2.1. Store and publish process

The T-DHT is a slightly modified version of a traditional DHT (e.g. Pastry, Chord or Tapestry). In a typical DHT when a user application needs to store the reference to an object, a hash calculation (e.g. SHA-1) is made on the object (or some object description) in order to get a unique object identifier. After that, a store packet is sent towards the node responsible of storing the previously calculated identifier. This identifier implicitly includes the routing path to the node responsible for storing the object reference. This process is a typical one in DHT systems and is implemented more or less in the same conceptual way in all the current DHT algorithms.

The difference between our system and a typical DHT lies on the storage of additional tag information, relative to the published object, into the node links. The tag information is inserted in node links in a two phase procedure named store and publish procedure. Subsequently, this previously inserted tag information will allow searching for the objects fulfilling a tag conjunction defined by the user queries.

As we can see in Fig. 1, in order to store and publish an object with hash value "Id" and his associated tags: "t1", "t2" and "t3", a two phase store and publish procedure is carried out.

First, a store operation is started. The T-DHT uses the object "Id" value to route the store packet across the overlay, in the same way a typical DHT does, until this packet reaches the node responsible for the storage of value "Id" (node N4). During the travel, each traversed node (N1, N2 and N3) stores the object tags "t1", "t2" and "t3" into the bloom filter (BF) associated with its output link. In a later search procedure, this information will allow knowing that an object tagged with "t1" and/or "t2" and/or "t3" can be located following that link.

The bloom filter data structure implements two operations: *insert (tag)* and *is_included (tag)* in a compact and efficient way. Since the bloom filter size is predetermined and constant at startup phase, we can guarantee the system scalability. A little trade-off could be expected during the search routing process due to the probability of false positive generation in bloom filters. This potential drawback is analyzed in the next section.

Finally, the store packet reaches N4, the node responsible for the storage of key "Id". Then, the node N4 starts

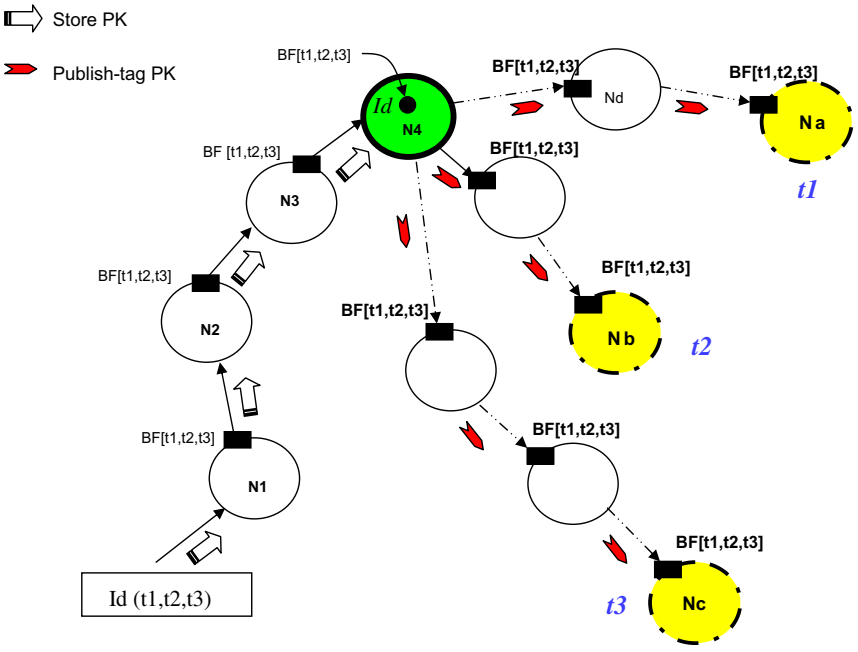



Fig. 1. Store and publish protocol.

the publish sub-phase. In this sub-phase we propagate to the network that the node N4 is storing a key with tags associated "t1", "t2" and "t3". In order to publish this information, the node N4 generates three *publish-tag* packets containing as destination the hash value of "t1", "t2" and "t3". The three *publish-tag* packets are routed towards the nodes Na, Nb and Nc, being Na, Nb and Nc the nodes responsible to store the key of tags "t1", "t2" and "t3", respectively. The purpose of these *publish-tag* packets is to store tag information across the paths N4-Na, N4-Nb and N4-Nc.

It should be noted that in this sub-phase the three tags are inserted in the input link of the destination node as opposed to the store sub-phase where the tags were inserted in the output link of the source node. This is done in this way because we are building something like a reverse path forwarding structure, and in this case, the target node (N4) is in the opposite path.

The whole process is based on DHT typical routing procedures, and so, we can guarantee scalability with at most O(Log($N$)) hops in both store and publish sub-phases. Also, the store process is fully balanced because we store into the nodes only the object key identifier. Besides, no popular tag hot spot storage problem can appear in the nodes because tag information is only stored in the bloom filters created with a fixed size. Only when publishing tag conjunctions including popular tags, some hot spot problems could appear in the links near the node responsible for storing the popular tag reference. Even in this scenario, T-DHT achieves an improved solution when compared with traditional inverted index solutions where the hot spot problem is not only located in links but also in nodes. Furthermore, analyzing experimental results, we did not discover any severe hot spot problem in the links near to nodes storing popular tag references during publishing phase.

### 2.2. Search and locate process

The search and locate process uses the tag information previously inserted into the node links in order to locate the objects that fulfill the tag conjunction proposed by a user query. In the first place, a typical DHT search process is started. During the travel of search packet, some traversed nodes can generate extra locate packets that will be relayed to other nodes following the tag information previously inserted in the bloom filters of their links. In parallel, the search packet will follow the DHT routing path as usual. We can see this protocol in Fig. 2.

Initially, a user application needs to locate some objects tagged with a tag set $\{t1, t2, \ldots, tn\}$. Therefore, the user application injects into the network a *search* packet from a node (e.g. node SA), including the tag information that the target object should fulfill (e.g. "t1", "t2" and "t3"). The destination of this *search* packet is the identifier obtained from the hashing of any of the tags, for example "t1".

In the worst case, the *search* packet will reach the node Na responsible for storing the object reference corresponding to the hash value of tag "t1". When the *search* packet reaches node Na, then the sub-phase *locate* starts. As we have seen in the former *store and publish* procedure, the node Na has tag information inserted in all its links. The node will scan its link bloom filters and it will realize the

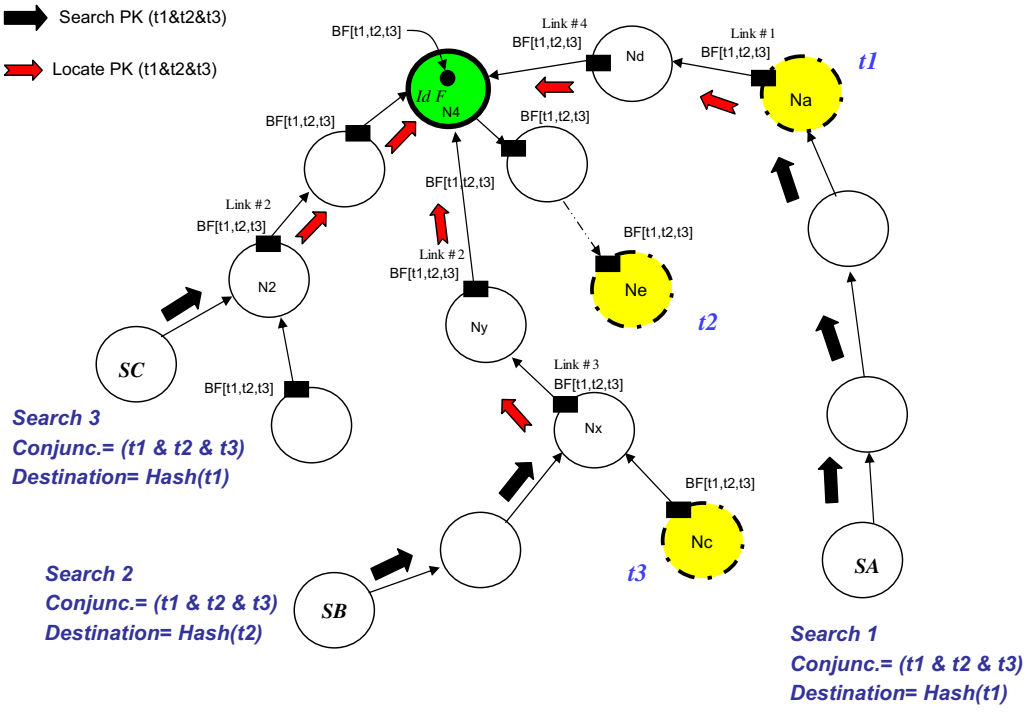



Fig. 2. Search and locate protocol.

link #1 indicates that the tags "t1", "t2" and "t3" are present following this link towards node Nd. Next, node Na will send a *locate* packet to node Nd. When node Nd receives the packet, it will scan all its link bloom filters in order to determine which of its links are candidates to resend the *locate* packet. In this case, link #4 is the candidate to retransmit the packet to node N4. Finally, when the locate packet arrives to node N4, this node will detect an object with identifier "id F" that fulfills the tag conjunction, and so, the node N4 will answer to the source node SA with the corresponding object reference. Obviously, a locate packet is never retransmitted to the link whereby the packet has been received.

More accurate scenarios are also possible. Consider node SB generating a search packet and addressing it towards the identifier obtained from the hashing of tag "t2". In this situation, the search packet reaches node Nx before arriving to node Ne (responsible for storing the key of tag "t2"). This node discovers, in its link #3, that tags "t1", "t2" and "t3" are located through this link. Hence, a *locate* packet will be sent to node Ny through link #3. Additionally, the *search* packet will be retransmitted from node Nx to node Nc which is responsible for storing tag "t2". Subsequently, the node Ny, after receiving the locate packet, will propagate this packet towards node N4 because the tag conjunction is included in the bloom filter of its link #2.

Another similar scenario could be represented by the search request generated by node SC sending its search packet towards Na (the responsible node of tag "t1" hash value). In this case, node N2 find link #1 as candidate to locate objects with tags "t1", "t2" and "t3", and generates the corresponding *locate* packet.

If a *locate* packet arrives to a node that does not detect any link containing all the tags requested, then the packet is discarded. This scenario is possibly due to a retransmission generated by a false positive in a former bloom filter. In other words, the false positives that bloom filters can generate, will inject false *locate* packets into the network. This potential overhead is not significant because with a high probability these packets will be discarded when arriving to the following nodes not fulfilling the requested tag conjunction.

Complex node configurations could generate packet loops due to locate packet retransmission. In order to avoid this problem, TTL and G-UID fields were added to T-DHT packet structure. The TTL field works as usual, decrementing one unit each time the packet arrives to a node, and discarding the packet when the TTL counter reaches zero. The globally unique identifier (G_UID) is a unique identifier associated to each packet that allows detecting packets reaching the node more than once time. When this happens the packet is discarded. In the simulation experiments we run, loop conditions were detected and avoided using exclusively the G-UID packet field. Nevertheless, the TTL field can also be used to modulate the exponential growth of locate packets when searching for popular tags. A deeper explanation of this feature is shown in Section 3.

As we can see from the first scenario, in the worst case $O(\mathrm{Log}(N))$ node hops are needed in order to reach the object fulfilling the tag conjunction. This is because (a) the first routing stage a search packet follows, is resolved using typical DHT routing procedures that are bounded by $O(\mathrm{Log}(N))$ node hops; and (b) the second routing stage a locate packet follows, has been previously generated in the opposite path from a DHT routing procedure, bounded again by $O(\mathrm{Log}(N))$.

From the scalability point of view, two types of packets will be traveling across the network nodes during the search process: search and locate packets. The search packets do not generate a scalability problem because only one packet at a time is been transmitted on the overlay, due to the fact the routing process done with this packet is a typical DHT routing process. However, during a search operation, several locate packets can be traveling on the network because, a new copy of a locate packet is generated each time the tag conjunction is included in some bloom filter. This situation could lead to a packet implosion if the tag conjunction contains very popular tags. In this case, the probability of a tag conjunction being included in bloom filter links is high, and so is the probability of generating new locate packets. Section 3 outlines possible solutions to cope with this scalability problem using the TTL field.

Finally, when starting a new search, T-DHT injects one *search* packet for each tag present in the tag conjunction set. However, when using very popular tags, the search packet addressed to a popular tag reference could generate hot spot problems around the node responsible for storing this popular tag reference. Some strategies can be implemented to diminish this problem: (a) using a very popular words dictionary, we can detect the potentially dangerous search packets addressed to the very popular tags, and so we can avoid injecting them into the system. The search procedure will work in the same way using only the other tags. The only constraint is that a very popular tag dictionary should be available; (b) a more conservative solution is to inject only a subset of all tags selecting them in a random way. In this way, only the selected tags would generate the corresponding search packets to be injected into the network. This strategy diminishes the frequency of very popular tags and so, the hot spot node problems. The current running version of T-DHT implements the later strategy and the run simulations have confirmed the nearly disappearance of hot spot problems when using very popular tag conjunctions.

## 3. Protocol extensions

The basic algorithm has been extended with some details in order to get a full operative prototype implementation.

The data persistence in T-DHT is assumed by the agent responsible for the object storage. Therefore, when a user

agent stores a tagged object, it must inject again this information periodically into the T-DHT network.

A tag and object caducity mechanism is implemented in order to automatically delete old tag and object information inserted previously into the T-DHT network and not refreshed adequately. Instead of using typical bloom filters to represent with one bit the occurrence or absence of a value, we use a slight modification that uses one byte to store the occurrence of a concrete value. The bloom filter new insert process does not set one bit as usual, but increments by one the byte field value. When we want to test for the occurrence of a value, we will test for a field value greater than zero. This mechanism allows keeping a certain history about how many set operations have been made in this value. Decrementing periodically the entire bloom filter fields in a node link, we achieve a simple aging mechanism. Thereby, a tag value that is not re-injected periodically (and so incrementing the corresponding bloom filter field values) sooner or later disappears from the bloom filter. This is because the aging procedure will decrement consecutively the bloom filter field until it reaches zero. Therefore the tag information inserted in this node link is deleted from T-DHT network. Also a typical local aging procedure is implemented to delete ancient object references from a T-DHT node.

The node failures are solved in a combined way by means of the data persistence refreshing method and the caducity mechanism previously explained. The former allows the insertion of objects and tags into the new nodes and links, and the later deletes the old tag references inserted into the node links. Transient data inconsistencies could be observed during the process described previously, but this model behavior is commonly tolerated in this kind of applications. For example, a Google search user expects to receive always more or less the same results, but tolerates certain *soft determinism* in their search operations. That is, the search result set obtained in two consecutive search operations is expected to be more or less the same, but it could be slightly different. Users of social network applications generally interact in a similar way, expecting a *soft determinism* in their search operations.

The update procedure was not described in the previous section, and it is implemented by means of a combination of two basic operations. First, deleting the old object attached tags, and second, inserting the new object attached tags. The delete packet format is similar to store packet. Inversely to the store packet, the delete packet erases the tag info from the node links traversed. If the old tags are not available when the delete operation is to be done, the caducity mechanism will eventually erase the tag information. However, during this transient period, there is the probability that the object could be located by means of the old and new tags simultaneously.

Finally, a stressing situation could appear when searching using only a popular tag due to the fact that a big amount of objects would fulfill the popular tag constraint. An exponential number of locate packets would be generated because the popular tag hash would be present in almost all the node links, and so a new locate packet would be created and sent to each node link. In order to bound the exponential growth of locate packets, a TTL heuristic has been added to the basic T-DHT protocol. The number of locate packets generated in a node is modulated adequately using as input parameters:

(a) Destination proximity. Since we do not know exactly the destination proximity, we will use the distance from the source by means of a node hop counter incremented each time a node receives a packet. As far as the packet is from the source, the locate packet growth is less problematic.
(b) Tag density in the current and previous node links. A big tag density will generate more locate packets, so if we know the current and previous tag density, we can estimate if the number of locate packets is going to be stressing.
(c) Tag popularity if available. If we know in advance the problematic tags we can apply a reduction ratio to them in order to diminish the problem.

## 4. Experiment results

To get realistic results we modeled our simulations with many parameters ranging from the queue size of physical link buffers to the popular tags used in social network applications.

In order to validate the system, we used a Jmyns, a packet-level, discrete event simulator, developed by us in order to evaluate our system. The simulator is a homemade, java coded discrete event simulator, similar in concept to Javasim, but focused and optimized to support a large number of peers and Internet-like topologies. Our simulator models Internet-like topologies, hosts, routers, physical links with different bandwidths and queuing delays, process time in routers, limited node buffer size and packet losses. The simulations were run on four different network topologies with 400,000 nodes (end hosts), 4500 internal routers and heterogeneous peer populations up to 10,000 peer nodes running T-DHT. These topologies were generated using the Georgia Tech [13] random graph generator with a transit-stub model. The physical link bandwidths were configured ranging from 4 Mbps (typical DSL home connection) to 40 Mbps in the internal links between backbone routers. The sizes of input packet queues of physical links were oversized in order to determine the maximum queue size distribution.

We implemented T-DHT on top of a slightly modified Pastry DHT overlay although other similar DHT overlays could be used in a similar way. The original implementation was coded in Java and targeted to run in Jmyns, our discrete event simulator. Currently we have coded again the T-DHT algorithm in a traditional way in order to be

executed on top a typical TCP/IP stack using sockets as transport interface.

Summarizing, the results obtained show a fully balanced T-DHT behavior in storage and search processes, independently of object tag distribution, even when using very popular tag distributions, in a range from 100 to 10,000 application nodes without generating hot spot problems.

### 4.1. Experiment setup

To model a real social network application environment we have designed four different types of experiments divided into two different scenarios: two experiments using tags following a popular tag distribution generated by means of a well known Zipf function; and the other two using tags following a nearly random pattern generation (actually a very soft Zipf distribution). All the tags are extracted from a real Spanish dictionary containing about 10,000 words. Since the tags are stored using the SHA-1 hash function to generate the corresponding identifier, a random identifier distribution is expected irrespective of the concrete dictionary content (e.g. Spanish or English dictionary).

In the two scenarios we stored object references tagged with seven different tags extracted from the dictionary. This procedure reflects the fact that people using social applications tend to describe their objects in a specific way in order to achieve, more easily, a successful location by other users.

In the two scenarios we use two different search procedures. The first search procedure is generated using only one tag. This situation reflects people searching in a very general way. When searching for popular tags this type of search tends to generate an exponential number of responses and the whole system should cope with this potential problem. The second search procedure is generated using three tags. This procedure represents the typical subset of users searching objects in a more specific way.

In the first scenario, we have associated more or less popularity to the dictionary words using a Zipf distribution. We have assumed intuitively that the search process uses a slightly more popular subset of tags than the storage process. The reason for this decision is that ordinary people tend to search for general popular tagged objects instead of searching for more specific ones. However during the storage process, people attach a mixture of popular and more specific tags to their objects to guarantee a subsequent accurate search, and so the tags used in the search process are generated by means of a narrower Zipf distribution. We can see the store and search Zipf distributions in Fig. 3.

In the second scenario we generated the store and search tags using a nearly random distribution (actually a soft Zipf distribution with a very long tail) in order to allow to compare performance results from a realistic scenario (Zipf tag distribution) versus a theoretical well balanced scenario (random tag distribution). The popular tag scenario currently generates hot spot problems due to the asymmetric nature of store and search operations.

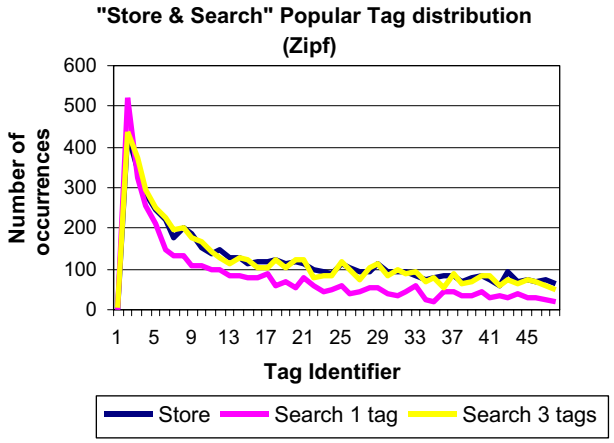



Fig. 3. Store and search Zipf tag distributions.

However, the nearly random tag distribution scenario generates a more balanced situation. In Fig. 4 we can see the store and search nearly random distributions.

The T-DHT system takes 9 ms to process each packet except for responses. Each response packet takes 1 ms to be processed. The average size of T-DHT packets is 500 bytes and the input packet queues of T-DHT agents are oversized in order to determine the maximum and average queue size distributions.

Each experiment injected 15,000 store operations tagged with seven different tags generated using the corresponding distribution as stated formerly, and 15,000 search operations tagged with one or three tags, depending on the concrete experiment. The store operations are initiated randomly by any of the 10,000 T-DHT agents. The set of 15,000 store packets is injected during 1.5 s (10,000 store_pkts/s). The search operations start 3 s before the store process is initiated, injecting 15,000 search packets during 0.75 s (20,000 search_pkts/s). Also the search operations are started randomly by any of the 10,000 T-DHT nodes.

### 4.2. Results

In the next paragraphs we will show the results obtained. We start showing the results related with local

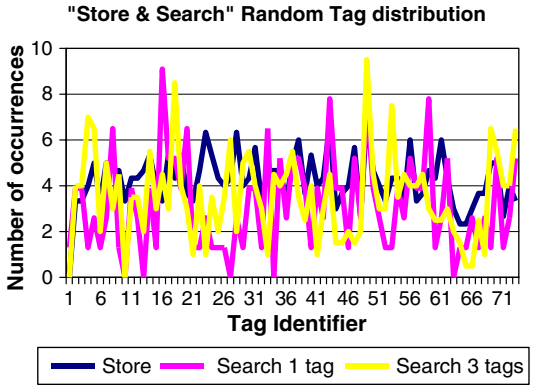



Fig. 4. Store and search random tag distributions.

nodes stress, concretely, the number of local store and search operations each T-DHT node must resolve individually. Next, we establish the round trip time values distribution of search queries in order to evaluate the scalability of T-DHT system coping with a great number of search queries. Subsequently, the distributions of the different types of T-DHT packets processed by the T-DHT nodes are shown. Finally, the distribution of the maximum size of input packet buffers in nodes is commented in order to assure the right scalability of internal T-DHT node resources.

Fig. 5 shows the cumulative distribution of the number of local store operations performed by the T-DHT nodes. This figure establishes the total number of local storage operations performed by T-DHT nodes along the whole experiment. It can be observed that the total storage effort (15,000 storage operations) is fully balanced among all the T-DHT nodes and therefore the system scalability is guaranteed. This result should not be surprising due to the fact that the storage procedure is performed in the same way a typical DHT does. That is, the object identifier determines the node responsible for storing the object reference. Since the object identifier is calculated by means of a hash function and this value has a random distribution due to the hash function properties, then the number of local storage operations shown in the four experiments follows a normal distribution. The mean value of these distributions is 1.5 objects per node as expected (15,000 objects distributed homogeneously among 10,000 nodes). Besides, in the worst case, a T-DHT node must store locally at most four object references during the whole experiment, and therefore, we can conclude that T-DHT avoids the hot spot appearance in storing nodes due to the fully balanced system behavior when storing objects. Finally, it should be noted that the four experiment curves overlap among them because the storage procedure is independent of the number and nature of the tags (popular or random) attached to the objects.

In Figs. 6 and 7 we show the cumulative distributions of search operations locally managed by T-DHT nodes during the whole experiment and during the 7th second (the maximum search load interval). A search operation is locally managed by a node when a search query is received and the search query condition is satisfied by a locally stored object. This situation generates a response packet towards the source node containing the object references fulfilling the received query.

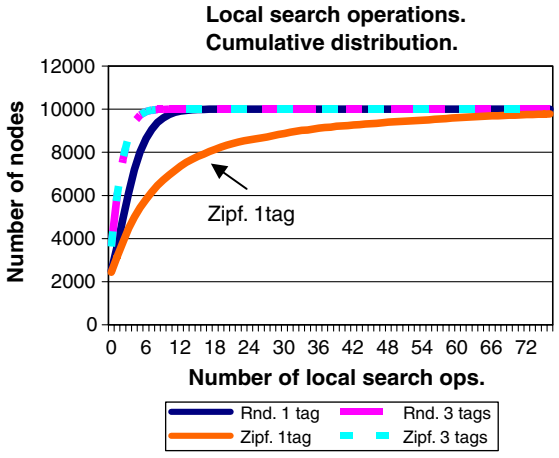



Fig. 6. Cumulative distribution of local search operations.

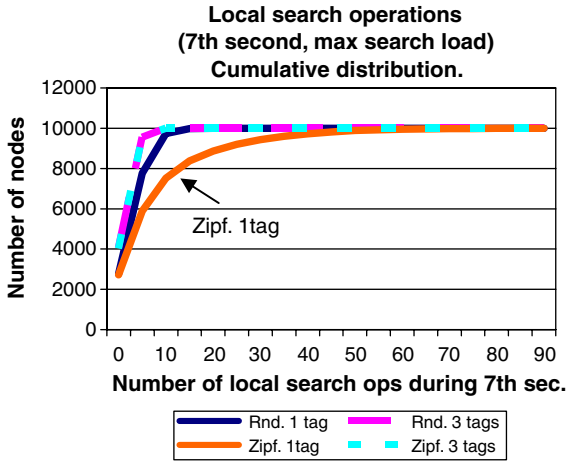



Fig. 7. Cumulative distribution of local search operations during 7th second.

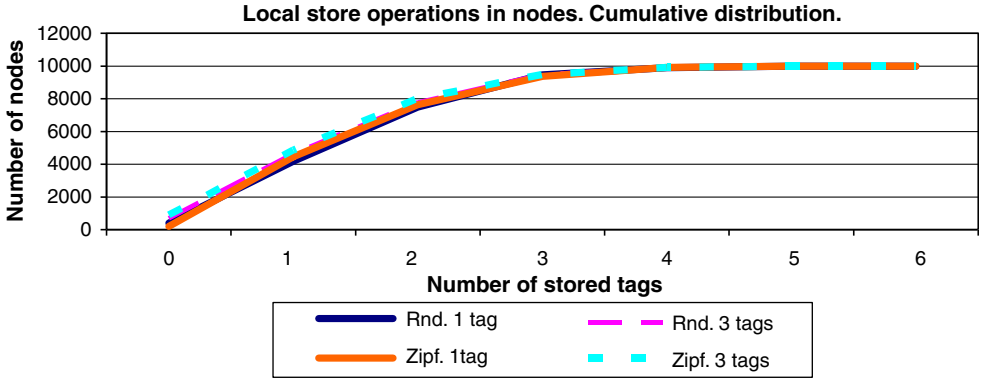



Fig. 5. Cumulative distribution of local store operations performed in nodes.

Analyzing Fig. 6, it is interesting to note that as it would be expected, the searching process with only one popular tag (zipf.1tag) forces more nodes to respond to more queries generating a slightly unbalanced overall behavior. Concretely, the last 20% of nodes must answer a range of queries from 20 to 80 throughout the whole simulation. This situation is produced because the search queries are routed mainly towards the nodes storing the popular tags. However, the searching process with only one tag, randomly distributed, shows a better scenario because the last 20% of the nodes must answer less queries (from 8 to 15 responses). In the other two scenarios, where the search query involves 3 tags (popular or random distributed), the last 20% of nodes must answer only a short and narrow interval of queries (from 3 to 5 responses). Besides, the 15,000 search operations, generated with a conjunction of three tags (popular or random distributed), produce at most five local search operations in the T-DHT nodes during the whole experiment. In other words, the search procedure does not generate hot spots in any T-DHT node even when searching popular tags. Finally, the two search distributions, "random with 3 tags" (rnd.3tags) and "popular with 3 tags" (zipf.3tags), have similar behavior, and so, the two curves overlap. This important result, already obtained during storage process, shows that T-DHT have a fully balanced behavior during the search procedure, independently of the distribution nature of searching tag sets.

Additionally, analyzing the 7th second distribution (Fig. 7), where values are taken when the search load is the highest along the whole simulation, the distribution curves are similar to those of Fig. 6. Logically the cumulated values are lower than Fig. 6 ones, because the aggregation is done only during 1 s.

Finally, the T-DHT search response behavior shown in Figs. 6 and 7 guarantees a fully balanced performance of T-DHT system response during the search process, even if the tag patterns used in the search process are totally unbalanced. Hence, we can say the T-DHT search response is fully balanced among all the participant nodes, and so, the system scalability is not compromised in this area due to the absence of hot spot in any T-DHT node, with the single exception of searches where the query only contains one popular tag. In other words, with the exception previously commented, a T-DHT system using an unbalanced data input (popular tagged objects) performs as well as a T-DHT system with a balanced data input (random tagged objects). This result is very relevant in the case of "zipf.3-tag" distributions, especially when comparing T-DHT with current solutions based on an inverted index, because in these solutions, hot spot problems emerge when using popular tags, even when the query is a tag conjunction (e.g. zipf.3tag).

In the next figures, we will generalize this result because it identifies a very important property of T-DHT: "when users search objects using a conjunction of tags and these tags follow a Zipf distribution, the T-DHT response is scalable, fully balanced and similar to the produced by searching objects using a conjunction of random distributed tags".

Fig. 8 identifies the round trip time distribution of search queries. All distributions are modeled nearly as narrow normal distributions with the exception of the "popular 1 tag" (zipf.1tag) distribution that presents a long right tail. This anomaly is explained taking into account that in this scenario we are searching with high probability the same object several times (i.e. the object tagged with a high popular tag), and so, the nodes storing this kind of objects are surely overloaded because nearly all requests are targeted to these nodes. The distribution narrowness in the other three curves guarantees a nearly deterministic global system response independently on the number of nodes and the absence of hot spots in this system response. Again the "rnd.3.tags" and "zipf.3.tags" distributions are identical, so their curves overlap, and hence they exhibit again a fully balanced system behavior despite the unbalanced tag popularity distribution.

Fig. 9 shows the distribution of the total number of packets processed by T-DHT nodes. As we stated formerly, searching popular tagged or random tagged objects by means of a small tag conjunction generates a nearly identical fully balanced T-DHT response. This important result can be shown, looking at "rnd.3tag" and "zipf.3tag" distributions, respectively. Again, the two distributions are nearly equal and they perform almost as normal distributions with a very low variance. The rnd1.tag also performs as a narrow normal distribution with a little greater mean value. This kind of data distributions assures fully balanced behavior and guarantees no node will receive a high number of packets to process. In the worst case, rnd3.tag and zipf.3tag indicate that the most loaded node received 90 packets and the expected mean value was 50 packets approximately per node. So, we can conclude that: (a) no node had to process a high number of packets and hence

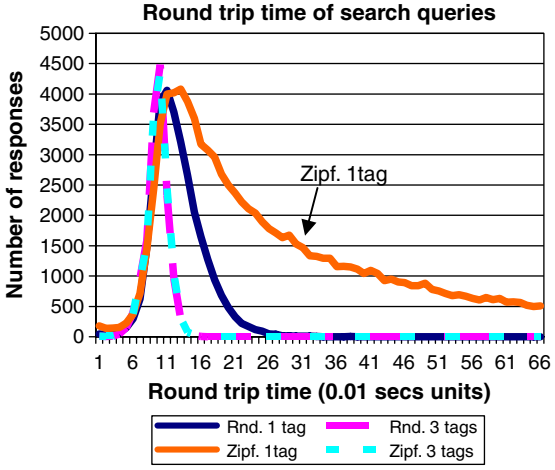



Fig. 8. Distribution of round trip time of search queries.

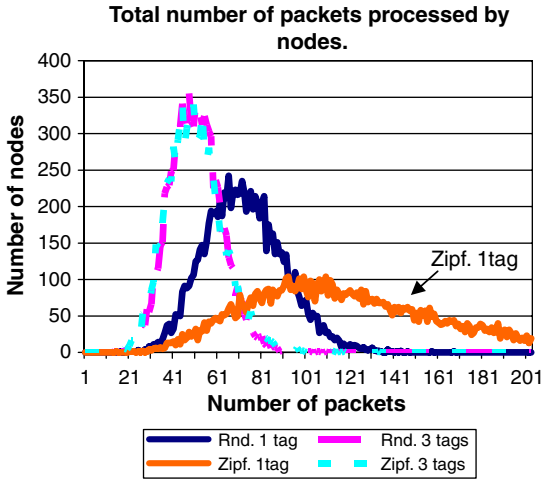



Fig. 9. Distribution of total number of packets processed by nodes.

no hot spot appeared; and (b) approximately 80% of the nodes processed a number of packets comprised in the range of [40...60], and therefore a balanced behavior has been achieved.

Finally, we can note again the unbalanced system behavior when searching popular tagged objects using only the popular tag (zipf.1tag).

In Figs. 10–14, we show the taxonomy of the values shown in Fig. 9. We can see similar behaviors when the different types of packets are analyzed, and so, the conclusions of Fig. 9 can be also applied to the next figures.

In Figs. 10 and 11 we show the packet distributions of store and publish packets, respectively. As expected the store and publish procedures are closely related with the traditional DHT storage process, and so, the four experiment distributions have nearly the same shape, and all of them are similar to a normal distribution with a low variance. From the values shown in these figures we can conclude that T-DHT nodes perform in a balanced and scalable way when processing store and publish packets, and no hot spots have appeared in any experiment.

Figs. 12–14 show the cumulative packet distributions of search and locate processes. The number of search packets processed by nodes (Fig. 12) scales adequately due to the fact that it is based on a deterministic DHT search. The

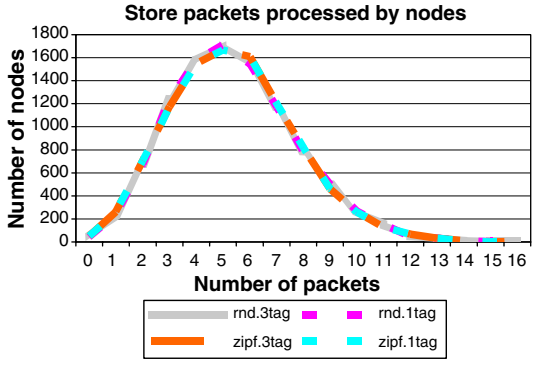



Fig. 10. Distribution of store packets processed by nodes.

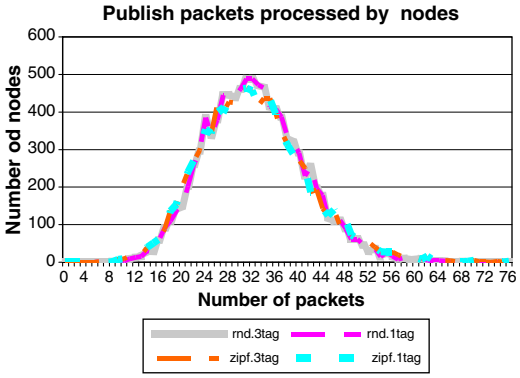



Fig. 11. Distribution of publish packets processed by nodes.

four distribution curves overlap showing a typical fully balanced DHT search behavior, where no hot spots have appeared. In the worst case, in the zipf.1tag distribution where searches are mainly targeted towards nodes responsible for storing popular tags, every T-DHT node must process on the average 2 search packets and at most 10 packets. These values show a balanced distribution of search packets, and the absence of significant hot spots when processing search packets. The other three experiments show even better values.

However the locate process (Figs. 13 and 14) presents a potential problematic situation when we search objects identified solely by one popular tag (Fig. 14). The problem arises because the very popular tags are present in almost all bloom filter links, since a great percentage of previously stored objects have been tagged with one of these very popular tags. Therefore, with a high probability, new locate packets are generated and transmitted every time one locate or search packet arrives at a node. In the limit, this situation resembles a flooding process. In our experiment, on the average, T-DHT nodes must process 60 locate packets in this scenario, but in the worst case, some of them can receive 170 packets. Anyway, in Figs. 13 and 14, we see that the other three distributions show better results. When using a conjunctive tag search (rnd.3tag and zipf.3tag), the maximum number of locate packets a node receives is 10, independently of the popularity of tags. Even when searching with only a random tag (rnd.1tag) the maximum value observed is 52 locate packets in a node.

Since T-DHT has been developed to work in a social networking scenario, we should consider worthless searching objects using only a very popular tag as search criteria. A similar scenario could be for example a Google search where the only keyword is a very popular one as "Disney". So, the potential negative impact of this problem probably should not appear. Furthermore, some solutions have been devised and described in the previous section to cope with this problem, as a TTL based forwarding heuristic, or a user application filter to avoid injecting "very popular only-one-tag" queries into the system.

Finally, Fig. 15 shows the distribution of maximum size of input packet buffers in nodes. This distribution is useful

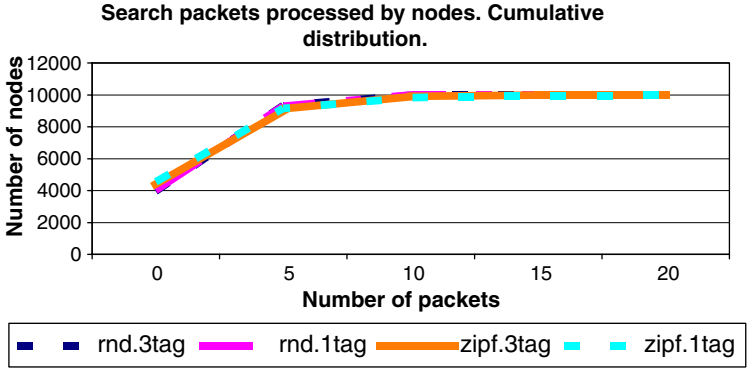



Fig. 12. Cumulative distribution of search packets.

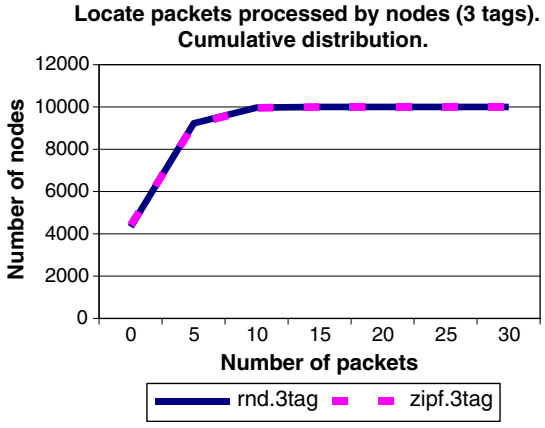



Fig. 13. Cumulative distribution of locate packets (3 tags).

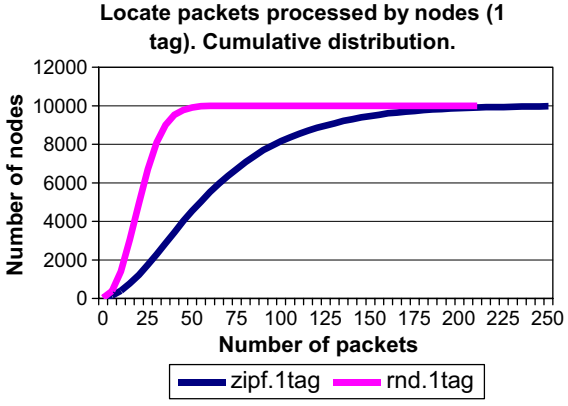



Fig. 14. Cumulative distribution of locate packets (1 tag).

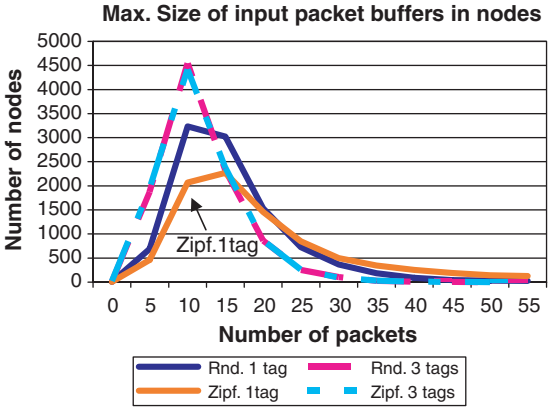



Fig. 15. Distribution of maximum size of input packet buffers in nodes.

when trying to identify hot spots in node links. A large number of packets, in an input queue, suggest a congested node generating a hot spot. Again, the "rnd.3tag" and "zipf.3tag" distributions are nearly the same distribution, and they can be represented by a normal distribution with a low variance. In these experiments, most T-DHT nodes only need on the average in the worst case a 10 packet long buffer in order to not discard any packet during all the developed simulations. Also, the most stressed T-DHT nodes only need a 27 packet long buffer to cope with the worst case scenario. Besides, the rnd.1tag and zipf1.tag distributions generate nearly normal distributions with similar values (expected value of 15 packets and 35 packets in the worst case).

The "1 tag" scenarios logically generate a right tail in their distributions due to the fact that some nodes will receive more packets because: (a) they are in the path of the frequently searched popular tagged object; or (b) they are the nodes responsible for storing the popular tagged objects. Solutions to diminish this anomaly have been exposed several times in this paper.

The analysis of these results is coincidental with the former ones: (a) T-DHT behaves in a fully balanced way (nearly normal distributions) when searching tag conjunctions despite the unbalanced nature of the search queries tagged with popular words; and (b) no hot spots have appeared (input buffer queues of links with a length of 35 packets in the worst case versus a mean value of 20 packets).

## 5. Related work

Searching objects in a social network using tags is a special case of the general problem of keyword search, where the keywords are user defined tags and the number of them in store and search processes is small. A lot of research

work in keyword search topic has been published in recent years, resulting in significantly improved techniques for conducting keyword search in structured and unstructured Peer-to-Peer overlays.

One of the first attempts to achieve keyword search in a distributed hash table was the distributed inverted index and researchers have proposed several solutions exploiting the initial concept [8,25–29] among others. Bhattacharjee et al. [9] proposed a solution based on view trees in order to process efficiently conjunctive queries, and mapping result caches into a tree structured overlay. A more general approach was followed by Huebsch proposing PIER [15], an Internet wide scope relational query processor using DHT as distributing mechanism. Also the IRISnet system [16] and Astrolabe [17] have similar goals but proposing a hierarchical approach instead of a balanced and distributed Peer-to-Peer approach.

Li et al. [18] discuss the feasibility of the keyword search of the Web using Peer-to-Peer architectures, and present existing and novel optimizations for Peer-to-Peer search based on DHTs, concluding the scalable solution will be a combination of several solutions and compromises related with quality of results and Peer-to-Peer system structure. Also, Yang et al. show in [19] that structured and unstructured systems use nearly the same bandwidth to process queries, and the structured overlays achieve the best system response when searching, but have a high cost of publishing and storing data.

Improved techniques for routing in unstructured overlays have appeared recently ([11,20–24] among others) trying to cope with scalability problems that unstructured overlays search algorithms suffer. Multiple random walks [20], probability routing tables [21], efficient super-peer networks [22], square root topologies instead of power-law ones [23], and hybrid search schemes (random walks and flooding) [24] are some of the proposed innovations in order to achieve system scalability and efficient search operations.

Up to now, the proposed algorithms in structured overlays are mainly based on optimizations of distributed inverted index. Reinolds and Vahdat [8] suggested sending the bloom filter content summaries among target peers, trying to avoid sending useless data not fulfilling keyword conjunction to the source node. However, the solution did not scale well because the search performance is not good enough as shown in [25]. Gnawali [26] proposed to index contents by means of keywords-sets instead of single keywords in order to minimize the data to be sent, sacrificing the efficiency of the publishing process due to the generation of an exponential growth of published and stored keyword combinations. In a similar approach, Yang and Ho [25] suggest to attach a content summary to each keyword in the inverted index, so that a query returns only a small number of results. Sankaralingam et al. [27] describe a distributed adaptation of Pagerank algorithm based on distributed inverted index, where each peer only transmits the top $x$% of a keyword hits to the next peer responsible for storage of next keyword. Anyway, all the proposed solutions, to a greater or lesser extent: (a) transmit to the source or to other peers, a lot of object references that do not fulfill the keyword conjunction, been discarded later, and so, wasting bandwidth resources; and (b) during the search of popular keywords, hot spots will be expected to appear in the nodes that store the popular keywords.

Finally, proposals [28,29] implement keyword queries as a vector that can be routed more or less efficiently in DHT. Tang et al. [28] propose pSearch, implementing a vector space model on a CAN DHT, where data and queries are represented as a vector where the search process is resolved routing the query vector in a multi-dimensional CAN overlay. Joung et al. [29] define a hypercube space that can be mapped to a DHT space. The objects in the hypercube are represented by a $r$-bit vector according to its keyword set, where the hashed keywords are mapped to a vector dimension, and the keyword search process is resolved traveling across a hypercube subset. The two proposals fail to solve the scalability problem in the search process when the keyword search is not an exact search but a superset search due to the exponential growth of the number of nodes it needs to visit to fulfill the query.

In summary, with the exception of T-DHT, the current solutions do not solve efficiently the keyword search problem in a scalable way, and avoiding hot spots appearance when popular keywords are used. The main ideas of T-DHT and basic protocol scenarios were presented previously in a short paper version in [14].

## 6. Conclusions and future work

In this paper we presented T-DHT, a tag indexed DHT to be used in social network applications for constructing a scalable search infrastructure, where the objects are mainly indexed by means of user defined keywords named tags. T-DHT is a hybrid unstructured–structured DHT Peer-to-Peer network overlay that allows storing social objects and searching for them through user defined tags, in a fully scalable, distributed and balanced approach. Since T-DHT is based on DHT design properties, the store and conjunctive tag search procedures are achieved in at most O(Log($N$)) node hops in order to reach the corresponding social object references.

The main contributions of T-DHT solution in the keyword search topic, are their clear advantages versus typical inverted index solutions adopted until now in Peer-to-Peer area: (a) we can search using conjunctive tag predicates without the need to generate explicit inverted index, because the object tag information is stored in a new and innovative way inside the node links; (b) there is no need to do a post-join operation in the source node in order to get the fulfilling results and, consequently, no huge amount of response traffic is generated towards the source node because T-DHT nodes only send to the source node the object references fulfilling the tag conjunction constraint; (c) no hot spot problems appear even when very popular

tags are attached to objects because the popular tag information is not centralized in any node, but it is spread among all the node links; (d) enforcing in the hot spot problem avoidance, when we search objects using conjunctions of popular tags, the T-DHT search response is fully balanced and scalable and performs identically to using conjunctions of random frequency tags as our simulations have demonstrated. This result has a great importance because T-DHT guarantees a fully balanced behavior despite the unbalanced nature of search queries.

Finally, although T-DHT has been devised to build a scalable social application infrastructure, it can be applied to solve typical Peer-to-Peer keyword search problems.

Future research can proceed along several paths. First, we should test T-DHT under extreme churn values in order to prove the system stability. Second, the current T-DHT socket implementation should be deployed in a real Peer-to-Peer environment to contrast the previously obtained simulations results, and to verify the system scalability and balanced behavior of T-DHT in a real test bed. And third, new algorithm optimizations should be devised in order to diminish the probability of hot spots appearance when using queries with only a single popular tag.

## References


[1] <http://en.wikipedia.org/wiki/Folksonomy>.

[2] P.A. Berstein, N. Goodman, E. Wong, C.L. Reeve, J.B. Rothnie Jr., Query processing in a system for distributed databases (SSD-1), in: Proceedings of the ACM Transaction of Database Systems, vol. 6, 1981.

[3] M. Stonebraker, P.M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, A. Yu, Mariposa: a wide-area distributed databse system, VLDB Journal (1996).

[4] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, Chord: a scalable Peer-to-Peer lookup service for Internet applications, in: ACM SIGCOMM, August 2001.

[5] S. Ratnasamy, P. Francis, et al., A scalable content-addressable network, in: Proceedings of the ACM SIGCOMM, August 2001.

[6] A. Rowstron, P. Druschel, Pastry: scalable, decentralized object location and routing for large-scale Peer-to-Peer systems, in: Eighteenth IFIP/ACM Conference on Distributed Systems Platforms (Middleware 2001), November 2001.

[7] B.Y. Zhao, J.D. Kubiatowicz, A.D. Joseph, Tapestry: an infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, Computer Science Division, April 2001.

[8] P. Reinolds, A. Vahdat, Efficient Peer-to-Peer keyword searching, in: Proceedings of the Fourth ACM/IFIPUSENIX International Middleware Conference, June 2003.

[9] B. Bhattacharjee, S. Chaawthe, V. Gopalakrishnan, P. Keleher, B. Silaghi, Efficient Peer-to-Peer searches using result-caching, in: Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS-03), February 2003.

[10] M. Ripeanu, Peer-to-Peer Architecture Case Study: Gnutella Network, Technical Report, University of Chicago, 2001.

[11] Y. Chawathe, S. Ratnasamy, L. Creslau, N. Lanham, S. Shenker, Making Gnutella-like P2P systems scalable, in: Proceedings of the ACM SIGCOMM'03, August 2003.

[12] B. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of ACM (July) (1970).

[13] E. Zegura, K. Calvert, S. Bhattacharjee, How to model an internetwork, in: Proceedings of the IEEE INFOCOM, April 1996.

[14] A. Mozo, J. Salvachua, Tag indexed DHT for scalable search infrastructure in social network applications, in: Proceedings of the IEEE Sixth International Conference on Peer-to-Peer Computing, September 2006.

[15] R. Huebsch, B. Chun, J.M. Hellerstein, B. Thau Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, A.R. Yumerefendi, The Architecture of PIER: an Internet-scale query processor, in: Proceedings of the CIDR Conference, January 2005.

[16] P.B. Gibbons, B. Karp, Y. Ke, S. Nath, S. Sesham, IrisNet: an architecture for a world-wide sensor web, IEEE Pervasive Computing (October) (2003).

[17] R. van Renesse, K.P. Birman, D. Dumitriu, W. Vogel, Scalable management and data mining using astrolabe, in: Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS) Cambridge, MA, March 2002.

[18] J. Li, B.T. Loo, J.M. Hellerstein, M.F. Kaashoek, D. Karger, R. Morris, On the feasibility of Peer-to-Peer web indexing and search, Lecture Notes in Computer Science, vol. 2735, Peer-to-Peer Systems II, 2003.

[19] Y. Yang, R. Dunlap, M. Rexroad, B.F. Cooper, Performance of full text search in structured and unstructured Peer-to-Peer systems, in: IEEE INFOCOM, 2006.

[20] Q. Lv, P. Cao, E. Cohen, K. Li, Shenker, Search and replication in unstructured Peer-to-Peer networks, in: Proceedings of the ACM International Conference on Supercomputing (ICS'02), June 2002.

[21] A. Kumar, J. Xu, E. Zegura, Efficient and scalable query routing for unstructured Peer-to-Peer networks, in: Proceedings of the INFOCOM, 2005.

[22] B. Yang, H. Garcia-Molina, Designing a super-peer network, in: Proceedings of the IEEE ICDE, 2003.

[23] B.F. Cooper, An optimal overlay topology for routing Peer-to-Peer searches, in: Proceedings of the ACM/IFIP/USENIX International Middleware Conference, 2005.

[24] C. Gkantsidis, M. Mihail, A. Saberi, Hybrid search schemes for unstructured Peer-to-Peer networks, in: IEEE INFOCOM, 2005.

[25] K. Yang, J. Ho, Proof: a novel DHT-based Peer-to-Peer search engine, IEICE Transactions on Communications Special Section on Networks Software E90-B (4) (2007).

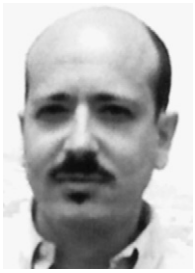
[26] O. Gnawali, A keyword set search system for Peer-to-Peer networks, Master's thesis, Massachusetts Institute of Technology, June 2002.

[27] K. Sankaralingam, M. Yalamanchi, S. Sethumadhavan, J.C. Browne, PageRank computation and keyword search on distributed systems and P2P networks, Journal of Grid Computing 1 (2003) 291–307.

[28] C. Tang, Z. Xu, S. Dwarkadas, Peer-to-Peer information retrieval using self-organizing semantic overlay networks. in: ACM SIGCOMM'04, Karlsruhe, Germany, August 2003.

[29] Y.J. Young, C.-T. Fang, 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 (ICDCS'05), IEEE Computer Society, 2005, pp. 339–348.


**Alberto Mozo** received a M.Sc. degree in Computer Science from Technical University of Madrid, Spain, in 1997 and Ph.D. degree from the same university in 2003. He is associate professor at Department of Computer Architecture and Technology at Technical University of Madrid. He has been involved in several European research projects related with collaborative e-learning environments. Current research activities include collaborative e-learning, Peer-to-Peer algorithms and Web 2.0 architectures.



**Joaquin Salvachua** received a M.Sc. degree in Telecommunications from Technical University of Madrid, Spain, in 1989 and Ph.D. degree from the same university in 1994. He was a lecturer at Technical University of Madrid from 1990 to 1995, when he was promoted to associate professor at the Department of Telematics Engineering. He has participated in over 12 European research projects with several publications at international magazines, books and conferences. Current research activities includes Advanced multimedia applications design, Human computer interaction, and multicast and P2P sharing with QoS protocol design. Also he works on the integration into the actual "Web 2.0" applications.