OpenKnowledge

FP6-027253

Initial Semantic Router

Spyros Kotoulas and Ronny Siebes

Faculty of Sciences, Vrije Universiteit Amsterdam, The Netherlands

Report Version: final
Report Preparation Date:
Classification: deliverable D2.2
Contract Start Date: 1.1.2006 Duration: 36 months
Project Co-ordinator: University of Edinburgh (David Robertson)
Partners: IIIA(CSIC) Barcelona
Vrije Universiteit Amsterdam
University of Edinburgh
KMI, Open University
University of Southampton
University of Trento

Adaptive routing in structured peer-to-peer overlays

Spyros Kotoulas and Ronny Siebes
Vrije Universiteit Amsterdam
Department of Computer Science
De Boelelaan 1081, 1081HV, Amsterdam
The Netherlands {kot,ronny}@few.vu.nl

Abstract

Finding content that is described by a large set of terms is a challenging problem in Peer-to-Peer (P2P) systems. One promising direction to solve it are Distributed Hash Tables (DHTs). The first DHT approaches were not able to efficiently store and retrieve the content because each term needs to be mapped to a key and routed to a peer that is responsible for it. Current solutions based on shared data-structures or on popularity-based metrics still have some disadvantages that may be tackled. We hope to contribute by this paper in two ways. First, we present an scalable and efficient multi-attribute routing algorithm in a structured DHT-like P2P network that adapts according to the popularity of the terms in the content description. Secondly, we have implemented this as a distributed discovery service for the OpenKnowledge system. For this implementation, we have performed a preliminary performance evaluation.

1 Introduction

Peer-to-Peer is a promising technology addressing some of the major challenges in modern distributed systems. First of all, it provides scalability through distribution of the deployment cost. Secondly, all peers have the same functionality, providing robustness by redundancy. Furthermore, it alleviates the problem of undisclosed and censored information; for example, a search engine might choose to censor specific information or rank information according not only to relevance, but according to advertisement income, just to name one. Finally, there is no guarantee of privacy, as the organization providing a specific service is free to log all and any information submitted by the user (for instance, search engines may correlate queries with IP addresses, domains etc). Notably, the currently most popular search engines are notoriously committed to the above. Peer-to-peer systems that rely on flooding have the most straightforward design and implementation. Nevertheless, they cannot scale, since the number of messages increases linearly with the size of the network, to preserve the same recall.

One promising direction to solve the scalability problem is the research on Distributed Hash Tables.

Distributed hash tables (DHTs) are currently seen as an important building block for peer-to-peer systems for storing and allocating content in a completely decentralized way [?]. This allows each node to function independently and collectively form the complete efficient search system without any central coordination. The general idea of DHTs is that each item shared on the network is hashed to a unique key, and that this key together with the content (or a pointer to it) are efficiently routed to the unique peer responsible for that key. In this way, each peer is responsible for storing the content (or a pointer to it) that is associated with the key. In principle, all DHT based systems provide two functionalities: store(key, object) storing an object identified by its key, and search(key) which returns the object (when it exists) from the peer whose network identifier is numerically closest to the key. The current systems based on DHTs provide these efficient key lookup and storage algorithms needing only O(log(N)) messages per search and storage, where N is the number of peers in the network. Although the current DHT solutions seem to deal well with automatic load balancing and robustness, there are still some challenges. One of them is multi-attribute searching and indexing. Especially when the number of keys for an item is large. For example, this research paper could be annotated with the following terms [`p2p, dht, search, experiments, rare-terms, ...`]. In current DHT solutions, each term would be a key, mapped with this document and stored in the network. One disadvantage is that it requires a lot of network usage and storage to put them all on the responsible peers. Also when you have multi-term queries, all terms in a query would be mapped to keys and routed to the responsible peers. The results would then be joined which is computationally expensive.

As a solution to the problem on managing large term sets and multi- term search, this work focussing on popularity-based approaches. The key idea is that popular content is easily available on the network due to high degree of replication. Therefore, we do not need to spend much effort on indexing it, in contrast to rare items. It is intuitively true, and experimentally verified[4], that for very popular items, we need no sophisticated routing, even a flooding approach would suffice. In addition, it has also been verified (looking at the results of previous research) and in [4], that for rare items, additional effort is required.

In [4], the authors suggest that for queries for common items, flooding queries is sufficient, while for rare items, DHT performs best. Research in the context of the PIER project [1] [1] also suggests a hybrid flooding/DHT mechanism (albeit with no efficient way to determine which items are rare). Indeed, for commons terms we are not interested in getting *all* results, if there are millions of them; a hundred would be enough. On the other hand, for rare terms, we are interested in *all* results. Nevertheless, most popularity-based approaches assume prior knowledge of which items are popular which is unrealistic. Our approach is to use statistical information, which is automatically calculated in a distributed way, to determine, on-the-fly, which terms are rare and which queries refer to them, and adapt the routing process accordingly.

Since this work is focussed on indexing before querying, we do not investigate caching techniques [3]. The methods proposed in this paper are orthogonal to these solutions and therefore it may be expected that both can benefit from each-other.

In the next section we show the naive traditional DHT approach and its fallacies. After that, in section 2 we show our approach. Section 3 shows our design, implementation and evaluation of the algorithm. We conclude our work in section 5.

2 Our approach

In our approach we exploit the structure in DHTs to extract interesting statistical information. As we mentioned in the introduction, the objects in our DHT are described by a set of terms, for example a keyword list. We call this set a *descriptor*. In our default implementation, these descriptors may be placed in *bins*, which are peers where one of the terms hashes to. Each of these bins(peers) will contain many (or all) descriptors for this specific term. We expect that this information alone is enough to calculate interesting statistical information. As a simple example, we can calculate the popularity of a feature or with which other features it frequently appears with. The statistical information can be used to optimize routing of queries and distribution of descriptors. In [4], the authors have observed and empirically evaluated that even though for popular items a flooding approach will return sufficient results, when it comes to rare items, additional effort should be made to provide for effective retrieval. By using the bins, we can easily find out how popular term is, so we can adjust the routing process and the descriptor placement in order to cater for rare items. Similarly, if two features tend to appear frequently together, we can save space and network traffic by placing the descriptor only on one of the two bins.

Now we show which settings we test in our experiments.

2.1 Different settings we test

Considering the number of design choices,it is impossible to perform an exhaustive search over all possible designs and strategies. Consequently, we made educated guesses on settings that may work well based on doing some prior experiments. We will leave testing of all possible techniques presented as future research. Therefore, following our analysis of existing and possible new approaches we have picked out a number of interesting settings to experiment on:

2.1.1 Setting 1 - Use the DHT as a distributed index

A simple p2p indexing system based on a DHT. Similar to the current indexing system of JXTA. The DHT is used as a distributed index and query answering is based on distributed joins.

Inserting Descriptions For every term $t_1...t_n$ the description, we store the tuple $[t, Description_{ID}]$ in the responsible peer. On a system with unlimited resources and no failures, this would lead to perfect recall.

Querying For a query $q(t_1...t_n)$, we retrieve all tuples with $t_1...t_n$. We perform a join to retrieve the relevant documents.

2.1.2 Setting 2 - Replicate whole term-set

Similar to the previous setting, but instead we insert the whole description instead of only the $[t, Description_ID]$ tuple.

Inserting Descriptions For every term $t_1...t_n$ the description, we store the tuple $[t_1...t_n, Description_ID]$ to the responsible peer. On a system with unlimited resources and no failures, this would lead to perfect recall. We need the same number of messages as in the first setting, but the size

[1] http://pier.cs.berkeley.edu

Algorithm 1 Setting 1

Require: A description d with terms $(t_1 \cdots t_n)$ and identifier id .
Ensure: q is stored.

1: **for** $i := 1$ to $i := n$ **do**
2: send($\{t_n, id\}$,P_{t_i})
3: **end for**

Require: A query q for terms $(t_1 \cdots t_n)$.
Ensure: q is forwarded.

1: **for** $i := 1$ to $i := n$ **do**
2: send(q,P_{t_1})
3: **end for**
4: $performlocaljoin$

of these messages is much greater (equal to the size of all terms in the description + the size of the ID).

Querying It is enough to choose only one of the terms from the query, and send the whole query to the peer responsible for that term. That peer can locally match the query with its stored tuples, and return results with 100% recall.

Algorithm 2 Setting 2

Require: A description d with terms $(t_1 \cdots t_n)$ and identifier id .
Ensure: q is stored.

1: **for** $i := 1$ to $i := n$ **do**
2: send($\{t_1 \cdots t_n, id\}$,P_{t_i})
3: **end for**

Require: A query q for terms $(t_1 \cdots t_n)$.
Ensure: q is forwarded.

1: send(q,P_{t_1})

2.1.3 Setting 3 - Replicate only on a subset

Although it can guarantee 100% recall with only 1 query message, setting 2 is inappropriate for descriptions with many terms (hundreds). The network bandwidth and the space required to send and store the hundreds of replicas would be prohibitive. We can use the statistical properties of descriptions to reduce the number of replicas required, on the expense of additional query messages.

Inserting descriptions A random subset of terms s is chosen, and the description is replicated on the peers responsible for these terms, in a similar fashion with Setting 2.

Querying Queries are forwarded to the peers responsible for each of the query terms. Results are returned as in setting 2. Note that this approach does not guarantee 100% recall. Nevertheless, it offers a trade-off between expensive insertion(and storage) and expensive retrieval.

Algorithm 3 Setting 3

Require: A description d with terms $(t_1 \cdots t_n)$ and identifier id . Let parameter k be the number of peers to replicate to.
Ensure: d is stored.

1: $s_{1 \cdots k} := picksubset(t_{1 \cdots n}, k)$
2: **for** $i := 1$ to $i := k$ **do**
3: send($\{s_1 \cdots s_k, id\}$,P_{t_i})
4: **end for**

Require: A query q for terms $(t_1 \cdots t_n)$.
Ensure: q is forwarded.

1: **for** $i := 1$ to $i := n$ **do**
2: send(q,P_{t_1})
3: **end for**

2.1.4 Setting 4 - Walk the descriptions

On systems with a Zipf distribution of descriptions, finding queries that lie in many peers is easy and straightforward; we can just flood queries to the network and the probability of encountering such descriptions is quite high. On the other hand, for rare descriptions, flooding would require a very large number of messages and a large amount of time, if we also consider traffic congestion caused by flooding. On the other hand, DHTs can guarantee efficient routing, but registering all terms of a description in a DHT is very costly. Ideally, we would like a system which uses a cheap mechanism to replicate common descriptions and a mechanism providing high recall for rare queries. The previous approaches (especially 1 and 2) can be configured for high recall, but for a cost that is not acceptable. In this setting, we will exploit the fact that a peer responsible for a term, contains much semantic information about this term, namely with which other terms it appears.

Inserting descriptions Peers forward descriptions to the peer responsible for term with the lowest frequency in their

descriptions. Although unintuitive at first glance, it has the following characteristics: First, rare terms are favored against common terms, since, by definition, common terms will appear more frequently. Secondly, descriptions will be "spread" across a wide semantic area (also see sections **??** and **??**). Descriptions are thus walked to peers responsible for diverse and rare terms.

Querying We can use a similar approach for querying. Initially, the query is routed to the peer responsible for a random term in the query set. If not enough results are found, it is routed to the peer that is responsible for the term that appears the least number of times in that peer's description. In the event that there are still not enough results, similar to setting 4, peers may forward the query to the peers responsible for the terms that co-occur the most with the query term.

Algorithm 4 Setting 4

Require: A description d with terms $(t_1 \cdots t_n)$ and identifier id . Let parameter $hops$ denote the maximum number of hops the description can be forwarded, originating peer set Pr, the description set of this peer D
Ensure: d is stored.

1: $t := t_m | \forall t, |D_t| > |D_{t_m}|$ *and* $P_t \notin Pr$
2: **if** $hops > 0$ **then**
3: hops:=hops-1
4: send(q,hops,Pr,P_{t_m})
5: **end if**

Require: A query q for terms $(t_1 \cdots t_n)$. Let parameter $hops$ denote the maximum number of hops the query can be forwarded, originating peer set Pr and the description set of this peer D.
Ensure: q is forwarded.

1: **for** $i := 1$ to $i := n$ **do**
2: send(q,P_{t_1},$this$)
3: **end for**

Require: A query q for terms $(t_1 \cdots t_n)$, originating peer set Pr, the description set of this peer D.
Ensure: q is forwarded.

1: **if** enough results found **then**
2: **return**
3: **else**
4: $t := t_m \in (t_1 \cdots t_n) | \forall t, |Dt| > |Dt_m|$ *and* $P_t \notin Pr$
5: $Pr := Pr \cup this$
6: send(q,P_{t_m},Pr)
7: **end if**

Setting	#Ins. Mess.	Ins. Mess Size	#Q. Mes.	#Ans. M.
Sett.1	n	1	O(n)	$\|D(t_1)\|+...+\|D(t_n)\|$
Sett.2	n	n	1	$\|D(t_1...t_n)\|$
Sett.3	s	n	O(n)	$\Omega\|D(t_1...t_n)\|$
Sett.4	hops	n	O(n)	$\Omega\|D(t_1...t_n)\|$

Table 1. Costs in terms of network traffic. We can see the costs associated with the basic operations of the discovery system, abiding to settings one and two.

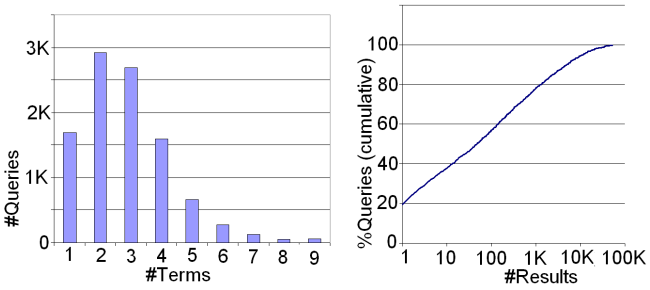

Figure 1. Left: Number of terms per query. Right: Number of results per query (cumulative). We can see that for approx. 50% of the queries, we have less than 50 results and for 30% of the queries, we have less than 4 results.

Now that we have shown the different settings, it is time to evaluate them.

3 Evaluation

rip some sections from proute/pnear/theses

3.1 Dataset

We have used the dataset developed for [2]. It was created by crawling a large number of real user queries from SearchSpy[2] and applying a natural language processing method on the results retrieved for these queries using Google[3], to get relevant descriptions. The input to our system was derived from the following:

- **Corpus** We have used a corpus of 260.000 descriptions. Each document was made up by a list of terms.

[2] http://www.infospace.com/info.xcite/searchspy
[3] http://www.google.com

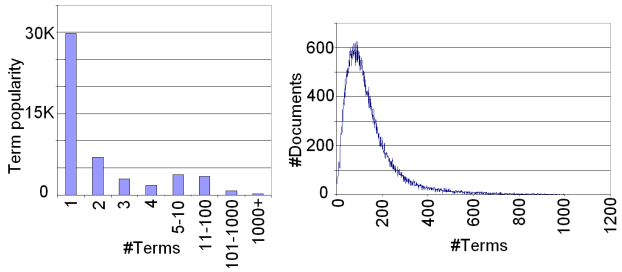

Figure 2. Left: Number of terms per query. Right: Distribution of the number of terms per document.

- **Descriptions** Out of these descriptions, we have selected a random set of 100.000 descriptions to use in our system. On average, each document contained approx. 104 terms (the distribution is shown in fig. **??**). The distribution of terms, as expected, follows a zipf-like distribution(fig **??** and **??**), we can see that more than half of all terms appear only 1 time, while 1 term appears in more than half of the descriptions (58204 times).

- **Queries** To generate queries, we have used the following method:

  1. Randomly pick the number of terms $|t|$ for the query using the probability distribution $0.16, 0.29, 0.26, 0.15, 0.07, 0.03, 0.01, 0.006, 0.006$ (fig. 2).
  2. Pick at random a description out of the Corpus.
  3. Pick $|t|$ terms (randomly using a uniform distribution) out of the chosen description and use them as the query terms.

  Note that not all queries yield results, because some of them are generated by documents from the Corpus that do not exist in the Descriptions.

In figure 2, we can see the number of answers per query; for most queries, there are fewer than 50 answers in the dataset.

3.2 Criteria

In order to evaluate our discovery system, we will use description *Recall*, *queries served*, and *query latency* . We consider *relevant* all documents that contain all terms in the query. To gain additional insight, we will always take into consideration the number of answers in the system, and limit them to a fixed threshold (50 answers). Thus, for our experiments, recall is defined as follows:

$$D_{Recall} = \frac{|D_{returned} \bigcap D_{relevant}|}{\min(|D_{relevant}|, 50)}$$

We will also perform a series of load tests, to measure the efficiency of our system. We will measure the total number of *queries served* and *query latency*:

```
bool isLocalPeerResponsible(String key);
void route(String key, Message msg);
void send(Node dest, Message msg);
void newMessage(Message msg); //Called
                              //when a new
                    //message is received
```

Table 2. The DHT API

3.3 Design, implementation and evaluation

Design Our system is based on a three-layer architecture:

- The bottom layer consists of a DHT implementation. We require only the basic functionality of a DHT, which is described in table **??**.
- The second layer consists of an object store and a distributed index supporting multi-attribute search. The distributed index relies on the algorithms described in 2.1.
- The third layer is application specific, in our case the OpenKnowledge service and peer discovery.

Implementation We have implemented our system using Java 1.5. For the bottom layer, we have used the FreePastry DHT implementation, version 2.0b[4]. The second layer are a translation of the algorithms in 2.1 in Java. The application on top is the discovery service of the OpenKnowledge platform [**?**]. It consists of a series of hosts collaborating to provide discovery of shared process-flows, services, software components and peers. The details of this application are out of the scope of this papers[5].

Testing environment We have used the DAS-2 distributed supercomputer[6] that provided computational power

[4] http://www.freepastry.org
[5] For more information, the reader is referred to http://www.openk.org
[6] http://www.cs.vu.nl/das2/

adequate to run a large number of instances of our distributed system. We have used one node on the DAS-2 as a bootstrap, which was used as an access point to the system for the rest. We have used Globus[] to start 500 instances of our system, which contact the bootstrap node, and self-organize according to the Pastry protocol. This process took less than 5 minutes. Next, instances published in parallel 200 descriptions each(100.000 in total). Finally, each instance made 100 queries and collected the results.

4 Results and discussion

The straightforward approaches 1 and 2 described in subsection 2.1 produced a very large number of messages which resulted in a very inefficient system, with no mentionable performance. Therefore, we have limited our evaluation to the two more efficient approaches, namely setting 3: random replication, and setting 4: replication according to rarest terms. Figure 3 shows the results of our experiments. We can see that even in a small overlay of 500 peers we gain a substantial increase in recall using the same number of messages. It is expected that when the network size grows, the recall of the random approach will deteriorate faster than that of setting 4. At the moment of writing this paper we are expecting that in the following weeks, the DAS-3 supercomputer[7] will be made available, providing up to 3.5 teraflops of processing power and 1TB of main memory. Furthermore, we have developed a grid-based simulator using the Ibis middleware[8]. With these tools we can verify if indeed it holds that when the network grows the difference in performance will be even larger in benefit of the rarity approach.

5 Conclusions

In this paper we propose an algorithm that reduces the scalability problem of multi-attribute search and annotations of objects in DHT networks. The intuition behind the algorithm is that indexes which store attribute sets for any object can deduce popularity of each individual attribute in its index. The algorithm automatically calculates attribute popularity and uses it to reduce the degree of replication which leads to a more equal distribution of attributes. We have implemented this algorithm in a real system which we tested by emulating 500 instances on the DAS-2 supercomputer. The results indicate that an algorithm that adapts routing according to attribute popularity outperforms an algorithm that either stores all attributes or a random subset. Future work lies in testing how this performance gain changes when the network size increases. Besides this, the robustness of our system towards a high peer churn rate needs to be tested.

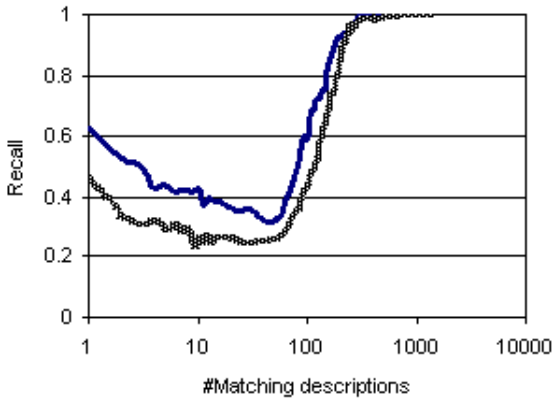

Figure 3. Description recall as a function of the number of matching descriptions per query.

Acknowledgements: This work has been supported by the FP6 OpenKnowledge project[9].

References

[1] B. T. Loo, R. Huebsch, I. Stoica, and J. M. Hellerstein. The case for a hybrid p2p search infrastructure, 2004.

[2] R. Siebes. pnear - combining content clustering and distributed hash tables. In *Proceedings of the IEEE'05 Workshop on Peer-to-Peer Knowledge Management.*, San Diego, CA, USA, July 2005.

[3] G. Skobeltsyn and K. Aberer. Distributed cache table: efficient query-driven processing of multi-term queries in p2p networks. In *P2PIR '06: Proceedings of the international workshop on Information retrieval in peer-to-peer networks*, pages 33–40, New York, NY, USA, 2006. ACM Press.

[4] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In *Proceedings of the IEEE INFOCOM conference*, San Fransisco, CA, USA, march 2003.

[7] http://www.cs.vu.nl/das3/

[8] http://www.cs.vu.nl/ibis

[9] http://www.openk.org/