Publish/Subscribe Functionality in IR Environments using Structured Overlay Networks*

Christos Tryfonopoulos Stratos Idreos Manolis Koubarakis

Dept. of Electronic and Computer Engineering
Technical University of Crete, GR73100 Chania, Crete, Greece
{trifon,sidraios,manolis}@intelligence.tuc.gr

ABSTRACT

We study the problem of offering publish/subscribe functionality on top of structured overlay networks using data models and languages from IR. We show how to achieve this by extending the distributed hash table Chord and present a detailed experimental evaluation of our proposals.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*information filtering*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*

General Terms

Algorithms, Performance

Keywords

publish/subscribe protocols,distributed hash tables,P2P overlay networks

1. INTRODUCTION

We are interested in the problem of *distributed resource sharing* in wide-area networks such as the Internet and the Web. In the architecture that we envision resources are annotated using attribute-value pairs, where value is of type text, and queried using constructs from Information Retrieval models. There are two kinds of basic functionality that we expect this architecture to offer: *information retrieval (IR)* and *publish/subscribe (pub/sub)*. In an IR scenario a user poses a *query* (e.g., "I am interested in papers on bio-informatics") and the system returns a list of pointers to matching resources. In a pub/sub scenario, also known as *information filtering (IF)*, a user posts a *subscription* (or *profile* or *continuous query*) to the system to receive notifications whenever certain events of interest take place (e.g., when a paper on bio-informatics becomes available).

In this paper we concentrate on the latter kind of functionality (pub/sub) and show how to provide it by extending the distributed hash table Chord [10]. *Distributed Hash Tables (DHTs)* are the second generation *structured* P2P overlay networks devised as a remedy for the known limitations of earlier P2P networks such as Napster and Gnutella.

We assume that publications and subscriptions will be expressed using a well-understood attribute-value model called $\mathcal{AWPS}$ in [6]. $\mathcal{AWPS}$ is based on *named attributes* with value *free text* interpreted under the Boolean and vector space models (VSM). The query language of $\mathcal{AWPS}$ allows Boolean combinations of comparisons A *op* v, where A is an attribute, v is a text value and *op* is one of the operators "equals", "contains" or "similar" ("equals" and "contains" are Boolean operators and "similar" is interpreted using the VSM or LSI model).

The contributions of this paper are the following. We present a set of protocols, collectively called *DHTrie*, that extend the Chord protocols with pub/sub functionality assuming that publications and subscriptions are expressed in the model $\mathcal{AWPS}$.

We evaluate DHTrie experimentally in a distributed digital library scenario with hundreds of thousands of nodes and millions of user profiles. Our experiments show that the DHTrie protocols are *scalable*: the number of messages it takes to publish a document and notify interested subscribers remains almost constant as the network grows. Moreover, the increase in message traffic shows little sensitivity to increase in document size. We demonstrate that simple data structures with only local information can make a big difference in a DHT environment: the routing table FCache manages to reduce network traffic by a factor of 4 in all the alternative methods we have studied.

Since probability distributions associated with publication and query elements are expected to be skewed in typical pub/sub scenarios, achieving a *balanced load* is an important problem. We briefly study an important case of load balancing for DHTrie and present a new algorithm which is also applicable to the standard DHT look-up problem.

The organization of the paper is as follows. Section 2 positions our paper with respect to related work. Section 3 introduces the model $\mathcal{AWPS}$ while Section 4 presents the

*This work was supported in part by project Evergrow. Christos Tryfonopoulos is partially supported by a Ph.D. fellowship from the program Heraclitus of the Greek Ministry of Education.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGIR'05, August 15–19, 2005, Salvador, Brazil.
Copyright 2005 ACM 1-59593-034-5/05/0008 ...$5.00.

DHTrie protocols. Section 5 presents the experimental evaluation of DHTrie. Section 6 studies the problem of load balancing. Finally, Section 7 concludes the paper.

2. RELATED WORK

The problems of IR and IF in P2P networks have recently received considerable attention. Here we only discuss the papers that are more relevant to our work. In [7] the authors study the problem of content-based retrieval in distributed digital libraries focusing on resource selection and document retrieval. They propose to use a 2-level hierarchical P2P network where digital libraries (called *leaf nodes*) cluster around directory nodes that form an unstructured P2P network in the second level of the hierarchy. In a more recent paper [8] the authors define the concept of neighborhood in hierarchical P2P networks and use this concept to devise a method for hub selection and ranking. The PlanetP [3] system uses an unstructured P2P network where nodes propagate Bloom filter summaries of their indices to the network using a gossiping algorithm. Each peer uses a variation of *tf/idf* to decide what nodes to contact to answer a query. In pSearch [12] the authors propose to use the CAN DHT and document semantic vectors (computed using LSI) to efficiently distribute document indices in a P2P network. In PIRS [16] the authors use an unstructured P2P network and careful propagation of metadata information to be able to answer queries in highly dynamic environments.

Early work on IF includes SIFT [14, 15] which uses the Boolean and vector space models, and InRoute [2] which is based on inference networks. Both of these systems are centralised although some issues related to distribution have been studied in SIFT [15]. Recently, a new generation of IF systems has tried to address the limitations imposed by centralized approaches by relying on ideas from P2P networks. The system P2P-DIET [4] (that builds on the earlier proposal DIAS [6]) is a retrieval and filtering system that uses the model $\mathcal{AWPS}$ and is implemented as an unstructured P2P network with routing techniques based on shortest paths and minimum-weight spanning trees. pFilter [11] is the closest system to the ideas presented in this paper. It uses a hierarchical extension of CAN [9] to filter unstructured documents and relies on multi-cast trees to notify subscribers. VSM and LSI can be used to match documents to user queries. By comparing pFilter with the proposals of this paper, we can see that we have a more expressive data model and query language, and do not need to maintain multi-cast trees to notify subscribers. However, the multi-cast trees of pFilter take into account network distance something that we do not consider at all in this paper. We also consider load balancing issues that are not studied in pFilter. Finally, regarding routing, it would be interesting to compare experimentally a hierarchical extension of our work with the hierarchical routing protocols of pFilter.

3. THE DATA MODEL $\mathcal{AWPS}$

We will use a well-understood attribute-value model, called $\mathcal{AWPS}$ in [6]. A *(resource) publication* is a set of attribute-value pairs (A, s), where A is a *named attribute*, s is a *text* value and all attributes are *distinct*. The following is an example of a publication:

$$\{\ (AUTHOR, \text{“}John\ Smith\text{”}),\\ (TITLE, \text{“}Information\ dissemination\ in\ P2P\ ...\text{”}),\\ (ABSTRACT, \text{“}In\ this\ paper\ we\ show\ that\ ...\text{”})\ \}$$

The query language of $\mathcal{AWPS}$ offers *equality*, *containment* and *similarity* operators on attribute values. The containment operator is interpreted under the Boolean model and enables the expression of Boolean and *word-proximity* queries. The similarity operator is defined as the cosine of the angle of two vectors corresponding to text values from a publication and a query. Vector representations of text values can be computed as usual using the VSM or LSI models (but only the VSM model has been used in our implementation and experiments).

Formally, a query is a conjunction of atomic queries of the form $A = s$, $A \sqsupseteq wp$ or $A \sim_k s$, where A is an attribute, s is a text value, wp is a conjunction of words and proximity formulas with only words as subformulas, and k is a *similarity threshold* i.e., a real number in the interval $[0, 1]$. Thus, queries can have two parts: a part interpreted under the Boolean model and a part interpreted under the VSM or LSI model. The following is an example of a query:

$$(AUTHOR = \text{“}John\ Smith\text{”}) \wedge\\ (TITLE \sqsupseteq P2P \wedge (information \prec_{[0,0]} alert)) \wedge\\ (ABSTRACT \sim_{0.7} \text{“}P2P\ architectures\ have\ been...\text{”})$$

This query requests resources that have *John Smith* as their author, and their title contains the word *P2P* and a word pattern where the word *information* is immediately followed by the word *alert*. Additionally, the resources should have an abstract similar to the text value *"P2P architectures have been ..."* with similarity greater than 0.7.

4. THE DHTRIE PROTOCOLS

We implement pub/sub functionality by a set of protocols called the *DHTrie protocols* (from the words DHT and trie). The DHTrie protocols use *two levels of indexing* to store queries submitted by clients. The first level corresponds to the partitioning of the global query index to different nodes using DHTs as the underlying infrastructure. Each node is responsible for a fraction of the submitted user queries through a mapping of attribute values to node identifiers. The DHT infrastructure is used to define the mapping scheme and also manages the routing of messages between different nodes. The set of protocols that regulate node interactions are described in the next sections.

The second level of our indexing mechanism is managed locally by each node and is used for indexing the user queries the node is responsible for. In this level, each node uses a hash table to index all the atomic queries contained in a complex query by using their attribute name as the key. For each atomic Boolean query the hash table points to a *trie*-like structure that exploits *common words* and a hash table that indexes text values in equalities as in [13]. Additionally for atomic VSM queries an inverted index for the most "significant" query words is used as in [15].

VSM relies on term frequencies (*tf*) and inverse document frequencies (*idf*) to compute the vector representation of a text value. The computation of *idf* in an IR or pub/sub scenario needs global statistical information. [3] has shown that in IR scenarios it is enough to have an approximation of the exact *idf* values. In [12] each peer uses a set of randomly chosen peers to collect such statistics and merge the results to create an approximation of the global *idf* values. These statistics are updated periodically using sampling. It is an open problem how to achieve this in a pub/sub scenario, although the ideas of [3, 12] are relevant in this context as well. We are currently working on this problem and expect to report our results in a future paper.

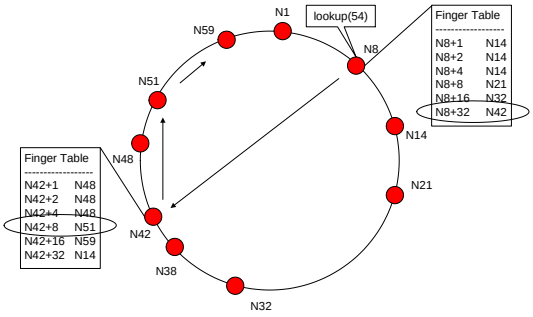

Figure 1: An example of a lookup operation over a Chord ring with m=6

In this paper we will focus on the first level of indexing and the protocols that regulate node interactions. The local indexing algorithms we use and their experimental evaluation are thoroughly discussed in [13, 15].

4.1 Mapping keys to nodes

We use an extension of the Chord DHT [10] to implement our network. Chord uses a variation of *consistent hashing* to map keys to nodes. In the consistent hashing scheme each node and data item is assigned a m-bit identifier where m should be large enough to diminish the possibility of different items hashing to the same identifier (a cryptographic hash function such as SHA-1 is used). The identifier of a node can be computed by hashing its IP address. For data items, we first decide a key and then hash it to obtain an identifier. For example, in a file-sharing application the name of the file can be the key (this is an application-specific decision). Identifiers are ordered in an *identifier circle (ring)* modulo 2^m i.e., from 0 to $2^m - 1$. Figure 1 shows an example of an identifier circle with 64 identifiers ($m = 6$) and 10 nodes.

Keys are mapped to nodes in the identifier circle as follows. Let H be the consistent hash function used. Key k is assigned to the first node which is equal or follows $H(k)$ clockwise in the identifier space. This node is called the *successor* node of identifier $H(k)$ and is denoted by $successor(H(k))$. We will often say that this node is *responsible* for key k. For example in the network shown in Figure 1, a key with identifier 30 would be stored at node $N32$. In fact node $N32$ would be responsible for all keys with identifiers in the interval $(21, 32]$.

If each node knows its successor, a query for locating the node responsible for a key k can always be answered in $O(N)$ steps where N is the number of nodes in the network. To improve this bound, Chord maintains at each node a routing table, called the *finger table*, with at most m entries. Each entry i in the finger table of node n, points to the first node s on the identifier circle that succeeds identifier $H(n)+2^{i-1}$. These nodes (i.e., $successor(H(n)+2^{i-1})$ for $1 \leq i \leq m$) are called the *fingers* of node n. Since fingers point at repeatedly doubling distances away from n, they can speed-up search for locating the node responsible for a key k. If the finger tables have size $O(\log N)$, then finding a successor of a node n can be done in $O(\log N)$ steps with high probability [10]. In [10] the details of the Chord protocols for node joins and leaves, stabilisation and fault-tolerance are provided. In the rest of this section we show how to extend Chord to implement our pub/sub functionality.

4.2 The subscription protocol

Let us assume that a node P wants to submit a query q containing *both* Boolean and VSM parts of the form:

$$A_1 = s_1 \ \wedge \ ... \ \wedge \ A_m = s_m \ \wedge$$
$$A_{m+1} \sqsupseteq wp_{m+1} \ \wedge \ ... \ \wedge A_n \sqsupseteq wp_n \ \wedge$$
$$A_{n+1} \sim_{a_{n+1}} s_{n+1} \ \wedge \ ... \ \wedge \ A_k \sim_{a_k} s_k$$

To do so, P randomly selects a single word w contained in any of the text values $s_1, \ldots, s_m$ or word patterns $wp_{m+1}, \ldots, wp_n$ and computes $H(w)$ to obtain the identifier of the node that will be responsible for query q. Then P creates message $\text{FwdQuery}(id(P), IP(P), qid(q), q)$, where $qid(q)$ is a unique query identifier assigned to q by P and $IP(P)$ is the IP address of P. This message is then forwarded in $O(logN)$ steps to the node with identifier $H(w)$ using the routing infrastructure of the DHT. This forwarding is done using the DHT lookup function to locate $successor(H(w))$, which is then directly contacted by P. Notice also that both $id(P)$ and $IP(P)$ need to be sent to the node that will store the query to facilitate notification delivery.

When P wants to submit a query q (i.e., with a VSM part only) of the form $A_{n+1} \sim_{a_1} s_1 \ \wedge \ ... \ \wedge \ A_n \sim_{a_n} s_n$, it sends q to *all* nodes in the list $L = \{H(w_j) : \ w_j \in D_1 \cup \cdots \cup D_n\}$, where $D_1, \ldots, D_n$ are the sets of *distinct* words in text values $s_1, \ldots, s_n$. In contrast to queries with a Boolean part, queries with a VSM part *only* need to be stored in all the nodes computed as above in order to ensure correctness in the filtering process. Sending the same message to more than one recipients is discussed in detail in the next section, where publication forwarding poses the same problem.

When a node P' receives a message FwdQuery containing q, it stores q using the second level of our indexing mechanism. P' uses a hash table to index all the atomic queries of q, using as key the attributes $A_1, \ldots, A_k$. To index each atomic query, three different data structures are also used: (i) a hash table for text values $s_1, \ldots, s_m$, (ii) a trie-like structure that exploits common words in word patterns $wp_{m+1}, \ldots, wp_n$, and (iii) an inverted index for the most "significant" words in text values $s_{n+1}, \ldots, s_k$. P' utilises these data structures at filtering time to find quickly all queries q that match an incoming publication p. This is done using an algorithm that combines algorithms BestFit-Trie [13] and SQI [15].

4.3 The publication protocol

When a node P wants to publish a resource, it first constructs a publication $p = \{(A_1, s_1), (A_2, s_2), \ldots, (A_n, s_n)\}$ (the resource description). Let $D_1, \ldots, D_n$ be the sets of *distinct* words in $s_1, \ldots, s_n$. Then publication p is propagated to *all* nodes with identifiers in the list

$$L = \{H(w_j) : \ w_j \in D_1 \cup \cdots \cup D_n\}.$$

The subscription protocol guarantees that L is a superset of the set of identifiers responsible for queries that match p.

The propagation of publication p in the DHT proceeds as follows. P removes duplicates from L and sorts it in ascending order clockwise starting from $id(P)$. This way we obtain at most as many identifiers as the distinct words in $D_1 \cup \cdots \cup D_n$, since a node may be responsible for more than one of the words contained in the document. Having obtained L, P creates a message $\text{FwdResource}(id(P), pid(p), p, L)$, where $pid(p)$ is a unique metadata identifier assigned to p by P, and sends it to node with identifier equal to $head(L)$ (the first element of L). This forwarding is done by the following

recursive method: message FwdResource is sent to a node P', where $id(P')$ is the greatest identifier contained in the finger table of P, for which $id(P') \leq head(L)$ holds.

Upon reception of a message FwdResource by a node P, $head(L)$ is checked. If $id(P) < head(L)$ then P just forwards the message as described in the previous paragraph. If $id(P) \geq head(L)$ then P makes a copy of the message, since this means that P is one of the intended recipients contained in list L (in other words P is responsible for key $head(L)$). Subsequently the publication part of this message is matched with the node's local query database using the algorithm mentioned in Section 4.2 and the appropriate subscribers are notified. Additionally list L is modified to L' in the following way. P deletes all elements of L that are smaller than $id(P)$ starting from $head(L)$, since all these elements have P as their intended recipient. In the new list L' that results from these deletions we have that $id(P) < head(L')$. This happens because in the general case L may contain more than one node identifiers that are managed by P (these identifiers are all located in ascending order at the beginning of L). Finally, P forwards the message to node with identifier $head(L')$.

The publication protocol essentially involves sending the same message to a group of other nodes, namely those that are responsible for the distinct words contained in the text values of the different attributes of p. The obvious way to handle this over Chord is to create h different lookup messages, where h is the number of different nodes to be contacted, and then locate the recipients of the message in an *iterative* fashion using $O(h \log N)$ messages. We have also implemented this algorithm for comparison purposes.

Once all matching queries have been retrieved from the database of a node P, notifications are sent to the appropriate nodes using their IP address associated with the query they submitted. If a node is not online at that time the notification message is sent to its successor. To utilise the network in a more efficient way, notifications can also be batched and sent to the subscribers when traffic is expected to be low.

4.4 Frequency Cache

In this section we introduce an additional routing table that is maintained in each node. This table, called *frequency cache (FCache)*, is used to reduce the cost of publishing a resource. Using the protocols described earlier, each node is responsible for handling queries that contain a specific word. When a resource r with h distinct words is published at node P, P needs to contact at most h other nodes to match the incoming resource against their local query databases. This procedure costs $O(h \log N)$ messages for each resource published at P. Since some of the words will be used more often at published resources, it is useful to store the IP addresses of the nodes that are responsible for queries containing these words. This allows P to reach in a single hop the nodes that are contacted more often.

FCache is a hash table used to associate each word that appears in a published document with a node IP address. FCache uses a word w as a key, and each FCache entry is a data structure that holds an IP address. Thus, whenever P needs to contact another node P' that is responsible for queries containing w, it searches its FCache. If FCache contains an entry for w, P can directly contact P' using the IP stored in its FCache. If w is not contained in FCache, P uses the standard DHT lookup protocol to locate P' and stores contact information in FCache for further reference. Using FCache the cost of processing a published resource p is reduced to $O(v + (h - v) \log N)$, where v is the number of words of p contained in FCache.

FCache entries are populated as follows. Each time a resource p is published at a node P, P contacts the nodes responsible for storing queries with words contained in p, as we described in Section 4.3. After this process is over, P knows the contact information (namely the IP address) of those nodes, and stores it to FCache along with the word each node is responsible for. After that, for each publication taking place at P, P maintains this routing information for the most frequent words contained in resources published to it. Notice that the construction and maintenance of FCache comes at no extra message cost and node routing information is discovered only when needed. In the experiments presented in the next section we discuss good choices for FCache size (Section 5.2).

The extra cost involved with FCache is possible cache misses because of network dynamicity. In an FCache miss the node needs to utilise the routing infrastructure at the cost of $O(logN)$ messages to locate a node. However the new contact information is used to update the FCache entry for future reference. Misses are most likely to occur for infrequent words, since nodes responsible for storing queries with frequent words will be contacted repeatedly.

5. EXPERIMENTAL EVALUATION

For our experiments we use 10426 documents downloaded from CiteSeer and used in [13]. The documents are research papers in the area of Neural Networks and we will refer to them as the NN corpus. Because no database of queries was available to us, our queries are synthetically generated by exploiting 2000 documents of the corpus. The remaining 8426 documents are used to generate publications.

Each query q has two parts: (i) a Boolean part which consists of at most 4 conjuncts that are atomic Boolean queries of the form $A \sqsupseteq wp$, where wp is a conjunction of at most 4 words or proximity formulas, and (ii) a VSM part which consists of at most 3 conjuncts of the form $A \sim_k s$, where s is a text value. Each atomic Boolean query of the form $A \sqsupseteq wp$ is generated using the methodology of [13]. We set A to be TITLE, AUTHORS, ABSTRACT or BODY with some probability. Then we set wp to a conjunction of words or proximity formulas obtained from technical terms mined from the document corpus. Each atomic VSM query of the form $A \sim_k s$ is generated as follows. We set A to be TITLE, ABSTRACT of BODY with some probability. Then we choose randomly a corpus document and set s equal to the TITLE, ABSTRACT or some part of the BODY field depending of our earlier choice of A. Finally, we set k to a value between $[0.3, 0.7]$ using the uniform distribution.

We have implemented and experimented with four variations of the DHTrie protocols. The first one, named It, utilises the iterative method in the publication protocol and does not use FCache. This algorithm was implemented mainly for comparison reasons. The second algorithm, named ItC, utilises again the iterative method and also an FCache, and is intended to show the effect of FCache when using the iterative method in the publication protocol. The third algorithm, named Re, utilises the recursive method in the publication protocol but does not use the FCache. Finally,

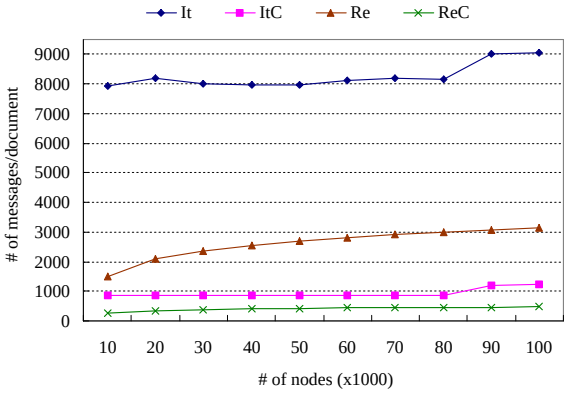

Figure 2: Performance for various network sizes

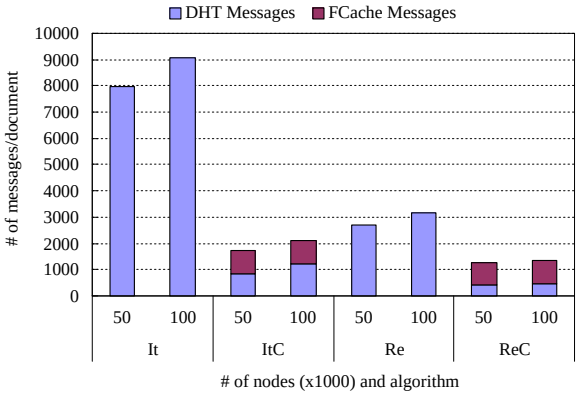

Figure 3: Total document processing cost

ReC uses the recursive method and FCache and shows a significant improvement in performance compared to the rest of the algorithms.

To carry out each experiment described in this section, we execute the following steps. Initially the network is set up by assigning keys to nodes. These keys are calculated using the SHA-1 cryptographic hash function and randomly created IP addresses and ports. After the network is set up, we create 5M user queries and distribute them among the nodes using the protocol described in Section 4.2. Once the queries are stored, we publish the corpus documents at different nodes and record the network activity.

5.1 Varying the Size of the Network

The first experiment that we conducted to evaluate our protocols targeted the performance of the algorithms under different network sizes. In this experiment we randomly selected 100 documents (with 5415 words average size) from the NN corpus and used them as incoming publications by randomly assigning each one to a publisher node. In each one of the 10 different runs, each document was assigned to a different node. Having published the documents, we recorded the total number of DHTrie messages generated by the network in order to match these documents against the posted user queries.

In Figure 2 the performance of the different algorithms in terms of DHTrie messages per document is shown. The main observation is that the number of messages generated by all the different algorithms to match the incoming documents against the user queries, remains relatively unaffected by the network size, mainly due to the routing infrastructure used. Thus, a document needs in the worst case 10% more messages for a 10 times larger network (see for example It for 10K and 100K nodes).

A second observation emerging from the graphs in Figure 2 is the effectiveness of the FCache independently of the message routing algorithm used. The use of FCache (with 30K entries) results in the reduction of messages sent using the routing infrastructure by more than 4 times in all cases (by using either the iterative or the recursive method). Notice also that using algorithm ReC reduces the DHTrie message cost of publishing a document of about 5500 words to only 500 messages for a network consisting of 100K nodes managing to process both Boolean and VSM queries.

Finally in Figure 3 we present the total cost for processing a single document in terms of message traffic for networks of 50K and 100K nodes. By total cost we mean the messages sent using information from FCache plus the messages sent using the DHTrie infrastructure. For algorithm ItC and a network of 50K nodes the DHTrie messages were about 50% of the total messages sent, whereas for a network of 100K nodes they were about 60%. On the other hand, for algorithm ReC the DHTrie messages were around 35% for both network sizes. The above observations show the importance of the recursive method and FCache in the reduction of the total document matching cost and in the relief in terms of messages of the DHTrie routing infrastructure.

5.2 Varying the FCache Size

The second experiment targeted the performance of the algorithms under different FCache sizes and studied the effect of FCache in the DHTrie utilisation. We used the document corpus as the training set for populating the FCache of the different nodes. We randomly selected a node p and published 10K documents to it. These publications resulted in the population of its FCache with the IP addresses of the nodes that are responsible for the most frequent words contained in the published documents, and served as a training set for the FCache. We then published 100 documents to p and limited the size of FCache to different values. Subsequently, we recorded the total number of messages generated by the network in order to match these documents against the stored user queries. Figure 4 shows the utilisation of the overlay network in messages per document as the size of FCache grows. The values shown are averaged over 10 runs with different nodes.

As it is shown in Figure 4 (left y-axis), the number of messages sent using the DHTrie routing infrastructure reduces quickly as the size of FCache increases to reach a state where the effect of an FCache increase causes no significant change in the number of messages (aroung 30K entries, the rightmost point in x-axis). Notice that the cost for each node to maintain an FCache consists only in storing this information in its local data store, namely about 24 bytes per entry (for storing the hash value of the word and the IP address of the node responsible for this word). Additionally the routing information of the FCache of node p depends only on the documents that get published to p, causing no additional

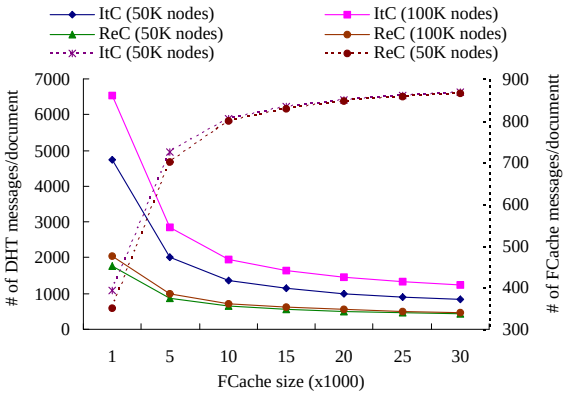

Figure 4: Performance for different FCache size

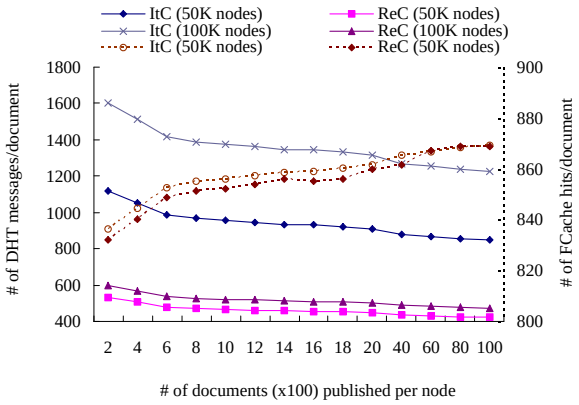

Figure 5: Different levels of FCache training

maintenance messages. The only extra cost involved with FCache is its update cost as nodes come and go from the network. This causes FCache entries to be outdated, costing more extra messages through the routing infrastructure to publish a resource. These extra messages though are sent only once, since the FCache field is updated when the new node responsible for the word with an outdated entry is located. From some initial measurements we found out that when 10% of the FCache entries are outdated, the message cost increase was no more than 4% showing that FCache is able to cope up with misses. Notice also that in the recursive method (algorithm *ReC*) the performance of FCache in different network sizes remains constant, whereas in the case of the iterative method (algorithm *ItC*) the performance deteriorates (50% more DHTrie messages per document for an 100% increase).

The right y-axis of Figure 4 shows the utilisation of FCache per document, showing again that after a threshold value (in our example around 30K entries, the rightmost point in x-axis) its effect is significantly reduced. This is also the reason that we chose 30K FCache entries as a baseline value for the rest of our experiments. We also observe that the number of messages sent using the FCache is about the same for both *ItC* and *ReC*, showing that FCache is equally utilised despite the algorithms used. For readability reasons we did not put the results for the 100K nodes network in the second y-axis, but they are similar to those presented.

5.3 Effect of FCache Training

In this experiment we measure the effect of FCache training to the message cost imposed to the network by the publication of a single document. We randomly selected a node P and trained P's FCache with a varying number of documents. Through this process the node was able to collect statistics about the most frequent words used in documents (published to it), and as a result it was able to populate its FCache with the appropriate pointers to frequently contacted nodes. Thus, for an FCache with 30K entries (the baseline value used in the experiments), the node would know the IP addresses for the nodes responsible for the 30K top most frequent words. We then published 100 documents (with 5415 words average document size) at P and recorded the message cost to match these documents against the stored user queries. The results shown in Figure 5 are averaged over 100 runs for different nodes to eliminate network topology effects.

Figure 5 shows that the performance of the different algorithms improves as more documents get published. Algorithm *ReC* seems less sensitive in this parameter, as the difference in the number of messages observed is about 100 messages for 50 times more documents (the leftmost and rightmost point in the x-axis), whereas *ItC* presents a difference of more than 300 messages. Additionally, *ReC* shows less sensitivity with respect to the network size, contrary to *ItC* that needs about 50% more messages. Finally, both *ItC* and *ReC* show a similar behaviour for the two network sizes we tested.

The right y-axis of Figure 5 shows the number of hits of FCache for different levels of FCache training. Notice that both algorithms have roughly the same number of hits for a network of 50K nodes, showing that FCache hits are not affected by the algorithm used. For readability reasons we did not put the results for the 100K nodes network in the second y-axis, but they are similar to those presented. Looking at the scale in the right y-axis, we can also see that the number of FCache hits shows only a slight improvement of around 4% for a 5000% increase in the number of documents used for training. This is attributed to the skewed nature of the data (documents) used to train the FCache. It is however important to note that even a small increase in FCache hits can significantly reduce message load (as it is already shown in the graphs of Figure 5), since every FCache hit, saves us from $O(\log N)$ DHTrie messages.

5.4 Varying the Document Size

Document (i.e., publication) size is an important parameter in the performance of our algorithms. This experiment targeted the performance of the different algorithms for varying document sizes. Each one of the bars in Figure 6 is an averaging of 100 documents, published at 1000 different nodes (in a network of 50K nodes in total) to normalise network topology effects. Figure 6 shows the message cost for publishing documents of varying size by using each one of the four different algorithms. Notice that the graph is truncated to a maximum of 5000 messages to show clearly the best performing algorithms.

Figure 6 shows that for small documents the use of the recursive method (contrary to FCache) does not improve

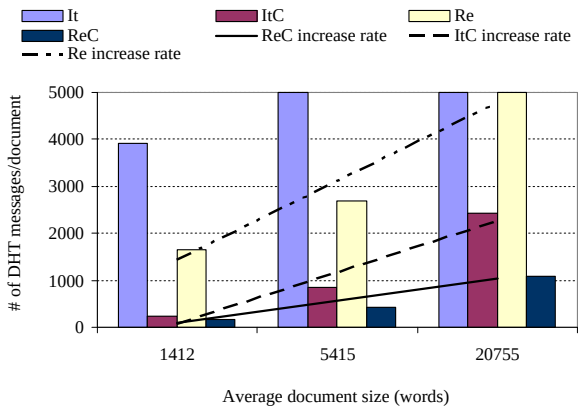

Figure 6: Performance for different document size

performance significantly, since algorithms *ItC* and *ReC* perform similarly. This is because for a large proportion of the words contained in small documents an FCache entry exists, thus needing a single message to reach the node responsible for queries that contain these words. The remaining words that are not listed in node's FCache use the DHTrie infrastructure, but their number is so small that we cannot observe significant differences in message cost. For large documents though, the use of the recursive method and FCache are shown to be significantly better than their counterparts, managing to process documents of average size of around 21K words by using around 1800 messages.

Finally another interesting observation emerging from Figure 6 is the increase rate for the different algorithms. The increase in message cost is linear to the document size with algorithm *ReC* presenting the smaller increase rate, thus showing a smaller sensitivity to document size.

6. LOAD BALANCING

In typical IR scenarios the probability distributions associated with documents and queries can be arbitrary and are typically skewed. For example, the frequency of occurrence of words in a document collection follows the Zipf distribution, subscriptions to an electronic journal might refer mostly to current hot topics while publications appearing in the same journal might reflect its established tradition etc. Thus, a key issue that arises when trying to partition the query space among the different nodes of a DHT in a pub/sub scenario is to achieve *load balancing*. In any pub/sub setting we can distinguish three types of node load: *query*, *routing* and *filtering*.

The *query load* of a node P is the number of queries stored at node P. The *routing load* of a node P is the number of messages that P has to forward due to the DHTrie protocols. Finally, the *filtering load* of a node P is the number of filtering requests (i.e., publications) that need to be processed at node P. Filtering is arguably the heaviest of the above tasks, since for each filtering request, a node has to search its local data store, retrieve the matching queries and notify interested subscribers. For this reason we choose to concentrate on balancing filtering load in this section. The filtering load of a node P depends on the number of words that hash to the interval of the identifier space owned by P and the frequency distribution of these words in published documents.

In the DHT literature, work on load balancing has recently concentrated on two particular problems: *address-space load balancing* and *item load balancing*. The former problem is how to partition the address-space of a DHT "evenly" among keys; it is typically solved by relying on consistent hashing and constructions such as virtual servers [10] or potential nodes [5]. In the latter problem, we have to balance load in the presence of data items with arbitrary load distributions [5, 1] as in our case.

We have implemented and evaluated a simple algorithm for distributing the filtering load evenly throughout the different nodes of the network. The algorithm is based on the well-known concept of *load-shedding*, where an overloaded node attempts to off-load work (i.e., filtering requests) to less loaded nodes. The algorithm is in fact applicable to the standard DHT look-up problem but here we utilize it in a pub/sub scenario. We will present a detailed analysis of the more general algorithm in a forthcoming paper.

The load balancing algorithm is as follows. Once a node P understands that it has become overloaded, it chooses the most frequent word w it is responsible for and a small integer k[1]. Then P contacts the nodes responsible for words wj for all j, $1 \leq j \leq k$ (wj is the concatenation of strings w and j) and asks them to be its replicas. Then P notifies the rest of the network about this change in responsibilities[2]. We Each node M that receives this message notes down the word w. Later on, if M has a new publication containing w, it breaks the filtering responsibility for w among P and k other nodes by concatenating a random number from 1 to k to the end of the w and using DHTrie to find the node responsible for this word. In this way the filtering responsibility of w for P is reduced by $k+1$ times (k new nodes plus P). We call $k+1$ the *split factor* (SF) in subsequent experiments.

In the experiments carried out in this section, a node P considers itself overloaded if it exceeds the threshold (T) of 10 filtering requests for the same word w in a time window of 100 document publications (in other words, if at least 10% of the published documents contain w). In a real network a node would not know how to define such a time window. In this case it could use sampling to estimate the average document publication rate, and thus be able to discover if it is doing more filtering work than other nodes.

The results of our experiments for the load balancing algorithm are shown in Figures 7 and 8. Figure 7 shows the average number of filtering requests received by each node in a time window for a period of 100 time windows. SF was set to 10 nodes and T was set to 10 requests/time window respectively. For readability purposes only the first 10K nodes (out of a total of 50K) are shown and the y-axis is truncated to 60 filtering requests (the highest point in the unbalanced case is 159 filtering requests). Notice that prior to the load balancing algorithm the first 3K nodes get a very large proportion of the filtering requests, whereas the rest of the network receives very few or no requests at all. On the contrary, after the load balancing algorithm is run, only a small amount of nodes receive more that 20 filtering requests, with the rest of the filtering load being distributed

[1]Currently we select the replica nodes randomly, but peer load or locality criteria could be used.

[2]For example, by piggy-backing the necessary information in DHTrie maintenance messages.

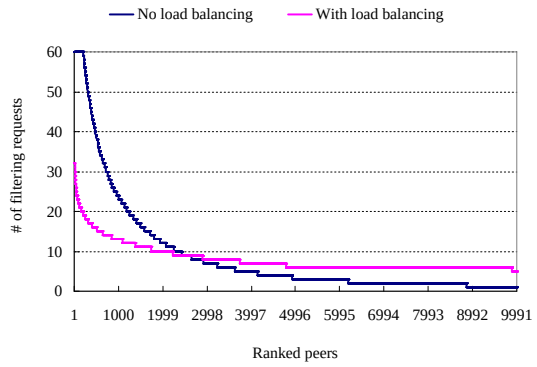

Figure 7: Average number for filtering requests

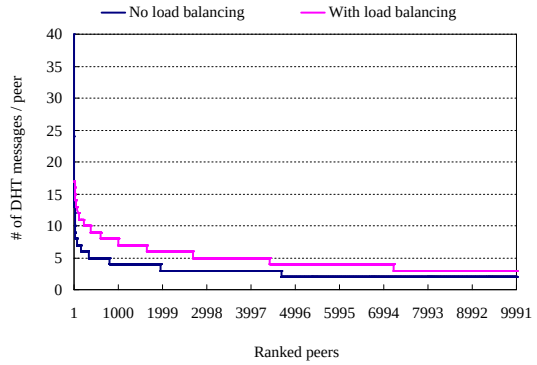

Figure 8: Routing load for the first 10K nodes

in a more uniform way among the nodes. We also experimented with different values for SF (20 and 30 nodes) and the T (20 and 30 requests/time window) but we did not observe significant differences in the load distribution.

Figure 8 shows the price we pay to achieve filtering load balancing in terms of message routing. In this graph we show the number of routing requests received by the first 10K nodes of our network. Notice that the number of messages needed per document increases significantly after the load balancing algorithm is run (we observed increases of as much as 80%). This increase is due to FCache misses occurring from the splitting of queries and filtering responsibilities. The increase in FCache misses causes a significant increase in DHTrie messages as it was expected (see Section 5.3), which is reduced when the FCache entries are updated. The important point however in Figure 8 is that the new load imposed on the network is uniformly distributed among the nodes and does not cause overloading in any group of nodes. The new load distribution follows closely the old one. This observation leads us to the conclusion that the load balancing algorithm shown here manages to efficiently distribute the filtering load among the nodes, while imposing a relatively small additional cost for routing purposes which in our scenario is considered an easier task to perform.

7. SUMMARY AND OUTLOOK

The evaluation of the DHTrie protocols revealed strengths and weaknesses of the different algorithms developed. In our experiments we showed that the DHTrie protocols are *scalable*: the number of messages it takes to publish a document remains almost constant as the network grows. Additionally we showed that the use of data structures that exploit local knowledge can significantly reduce network traffic (up to a factor of 4), with little overhead in training. Network traffic also presents little sensitivity to document size when FCache is utilised. Finally Section 6 presents a load balancing algorithm that trades message traffic for balance in the peer load. Distributed IR as studied in this paper can benefit from techniques of traditional IR, distributed systems and networking (especially P2P networks), databases and distributed AI. With this perspective in mind our current work concentrates on two open problems: algorithms for computing idf in distributed IR and pub/sub environments (Section 4), and implementing the load balancing algorithm of Section 6 for the standard DHT look-up problem and comparing it with related work.

8. REFERENCES

[1] K. Aberer, A. Datta, and M. Hauswirth. Multifaceted Simultaneous Load Balancing in DHT-based P2P systems: A new game with old balls and bins. Technical Report IC/2004/23, EPFL, 2004.

[2] J.P. Callan. Document Filtering With Inference Networks. In *Proc. of ACM SIGIR*, 1996.

[3] F.M. Cuenca-Acuna and T.D. Nguyen. Text-Based Content Search and Retrieval in Ad-hoc P2P Communities. In *Networking 2002 Workshops*, 2002.

[4] S. Idreos, M. Koubarakis, and C. Tryfonopoulos. P2P-DIET: An Extensible P2P Service that Unifies Ad-hoc and Continuous Querying in Super-Peer Networks. In *Proc. of SIGMOD*, 2004. Demo paper.

[5] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proc. of SPAA*, 2004.

[6] M. Koubarakis, T. Koutris, P. Raftopoulou, and C. Tryfonopoulos. Information Alert in Distributed Digital Libraries: The Models, Languages and Architecture of DIAS. In *Proc. of ECDL*, 2002.

[7] J. Lu and J. Callan. Content-based retrieval in hybrid peer-to-peer networks. In *Proc. of CIKM*, 2003.

[8] J. Lu and J. Callan. Federated search of text-based digital libraries in hierarchical peer-to-peer networks. In *Proc. of ECIR*, 2005. (to appear).

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[10] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, 2001.

[11] C. Tang and Z. Xu. pFilter: Global Information Filtering and Dissemination Using Structured Overlays. In *Proc. of FTDCS*, 2003.

[12] C. Tang, Z. Xu, and M. Mahalingam. pSearch: Information Retrieval in Structured Overlays. In *Proc. of HotNets-I '02*.

[13] C. Tryfonopoulos, M. Koubarakis, and Y. Drougas. Filtering Algorithms for Information Retrieval Models with Named Attributes and Proximity Operators. In *Proc. of ACM SIGIR*, 2004.

[14] T.W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM TODS*, 19(2):332–364, 1994.

[15] T.W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM TODS*, 24(4):529–565, 1999.

[16] W.G. Yee and O. Frieder. The Design of PIRS, a Peer-to-Peer Information Retrieval System. In *Proc. of DBISP2P*, 2004.