Implementing Publish/Subscribe Systems with Languages from Information Retrieval on Top of Structured Overlay Networks*

Christos Tryfonopoulos† Manolis Koubarakis

Dept. of Electronic and Computer Engineering,
Technical University of Crete, 73100 Chania, Crete, Greece
{trifon,manolis}@intelligence.tuc.gr

Abstract

We study the problem of distributed resource sharing in wide-area networks such as the Internet and the Web. The architecture that we envision supports both query and publish/subscribe functionality using data models and languages from Information Retrieval. We propose to approach this problem using ideas from self-organized overlay networks and especially distributed hash tables like Chord. This paper concentrates only on how to offer the envisaged publish/subscribe functionality and discusses our on-going work.

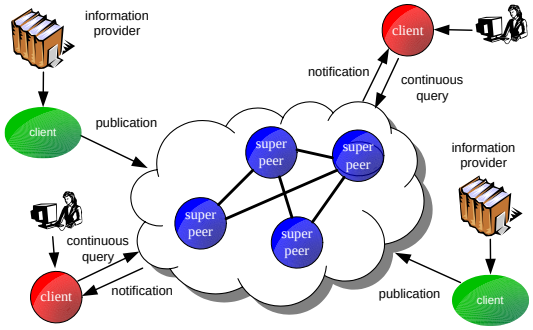

Figure 1: An architecture for distributed resource sharing

1 Introduction

We study the problem of *distributed resource sharing* in wide-area networks such as the Internet and the Web. Figure 1 presents a high-level architecture that has been adopted from our previous work [9, 6, 11, 4] and captures the problem to be solved successfully. There are two kinds of basic functionality that we expect this architecture to offer:

- *One-time querying*: a user utilizes his *client* to pose a *query* (e.g., "I want papers on self-organization") and the system returns a list of pointers to matching resources owned by other clients in the network.
- *Publish/subscribe (pub/sub)*: In a pub/sub scenario, a user posts a *continuous query* to the system to receive a notification whenever certain *events* of interest take place (e.g., when a paper on self-organization becomes available).

Our first solution to this problem is the system P2P-DIET [9, 6, 11, 4] which is now fully implemented and available from `http://www.intelligence.tuc.gr/p2pdiet`. In this paper we build on the lessons learned from P2P-DIET and show how to solve the problem of building pub/sub functionality in the architecture of Figure 1 using ideas from self-organized overlay networks such as *distributed hash tables* (DHTs).[1]

In our architecture we have two kinds of nodes: *super-peers* and *clients*. All super-peers are equal and have the same responsibilities, thus the super-peer subnetwork is a *pure* P2P network. Each super-peer serves a fraction of the clients. It is very easy to modify our proposal to work in the case of

*This work is partially supported by Integrated Project Evergrow (Contract No 001935) funded by the Complex Systems initiative of the FP6/IST/FET Programme of the European Commission.

†The author is partially supported by a Ph.D. fellowship from the program Heraclitus of the Greek Ministry of Education.

[1] P2P-DIET did not utilize DHTs; it followed the pub/sub system SIENA [3] and used arbitrary graphs and standard network routing algorithms based on shortest paths and minimum weight spanning trees.

pure P2P networks where all nodes are equal.

We will use the the data model $\mathcal{AWP}$ inspired from Information Retrieval for specifying queries and resource metadata [9]. For the purposes of this paper, 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:

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

The query language of $\mathcal{AWP}$ offers *Boolean* and *word proximity operators* on attribute values. The following is an example of a conjunctive $\mathcal{AWP}$ query:

$$AUTHOR = \text{“}John\ Smith\text{”} \wedge$$
$$TITLE \sqsupseteq p2p \wedge (information \prec_{[0,0]} dissemination)$$

The 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 *dissemination*.

We implement a distributed resource sharing by a set of protocols called the *DHTrie protocols* (from the words DHT and trie). The DHTrie protocols use *three levels of indexing* to store continuous queries submitted by clients. The first level corresponds to the partitioning of the global query index to different super-peers using DHTs as the underlying infrastructure. Each super-peer is responsible for a fraction of the submitted user queries through a mapping of attribute-value combinations to super-peer identifiers. The DHT infrastructure is used to define the mapping scheme and also manages the routing of messages between different super-peers. The set of protocols that regulate peer interactions are described in Section 2.

The other two levels of our indexing mechanism are managed locally by each one of the super-peers, as they are used for indexing the user queries that a peer is responsible for. In the second level each super-peer uses a hash table to index the attributes contained in a query, whereas in the third level a *trie*-like structure that exploits *common words* in atomic queries is utilised. The algorithm and the experimental evaluation are briefly described in Section 3.

2 The DHTrie protocols

In this section we describe DHTrie, a set of protocols that allow the partitioning of the global query index among the super-peers, using a DHT infrastructure.

2.1 Mapping keys to super-peers

We use a Chord-like DHT to implement our super-peer network. Chord [10] uses consistent hashing to map keys to nodes. Each node and data item is assigned an k-bit identifier, where k should be large enough to avoid the possibility of different items hashing to the same identifier. Identifiers can be thought of as being placed on a circle from 0 to $2^k - 1$, called the *identifier circle* or *Chord ring.* Data items are mapped to nodes in the Chord ring as follows. A new data item r is stored at the node with identifier $H(r)$ if this node exists, given that H is the hash function used. Alternatively, r is stored at the node whose identifier is the first identifier *clockwise* in the Chord ring starting from $H(r)$. This node is called the *successor* of node $H(r)$ and is denoted by $successor(H(r))$. We will say that this node is *responsible* for data item r. Node identifiers are assigned to nodes by hashing their respective IP addresses using a cryptographic hash function. When a node joins the Chord ring, its predecessor finds out that a new node has joined and makes this node responsible for data items hashing in identifiers between itself and the new node. Discovering that a node has joined is achieved through a stabilisation protocol that every node runs periodically. The idea behind stabilisation is to keep a node informed about its immediate successor in the Chord ring.

2.2 Subscribing with a continuous query

Let us assume that a client C wants to submit a continuous query q of the form $A_1 = s_1 \wedge ... \wedge A_m = s_m \wedge A_{m+1} \sqsupseteq wp_{m+1} \wedge ... \wedge A_n \sqsupseteq wp_n$. C contacts a super-peer S (its *access point*) and sends it a message SubmitCQuery$(id(C), q)$, where $id(C)$ is a unique identifier assigned to C by S in their first communication. When S receives q, it selects a random attribute A_i, $1 \leq i \leq n$ contained in q and a random word w_j from text value s_i or word pattern wp_i (depending on what kind of atomic formula of query q attribute A_i appears in). Then S forms the concatenation $A_i w_j$ of strings A_i and w_j and computes $H(A_i w_j)$ to obtain a super-peer identifier. Finally, S creates message FwdCQuery$(id(S), id(q), q)$ and forwards it to super-

peer with identifier $H(A_i w_j)$ using the routing infrastructure of the DHT.

When a super-peer receives a message FwdCQuery containing q, it inserts q in its local data structures using the insertion algorithm of BestFitTrie described briefly in Section 3 and also in [11, 7].

2.3 Publishing a resource

When client C wants to publish a resource, it constructs a publication p of the form $\{(A_1, s_1), (A_2, s_2), \ldots, (A_n, s_n)\}$, it contacts a super-peer S and sends S a message PubResource$(id(C), p)$. When S receives p, it computes a list of super-peer identifiers that are provably a superset of the set of super-peer identifiers responsible for queries that match p. This list is computed as follows. For every attribute A_i, $1 \leq i \leq n$ in p, and every word w_j in s_i, S computes $H(A_i w_j)$ to obtain a list of super-peer identifiers that, according to the DHT mapping function, store continuous queries containing word w_j in the respective text value s_i or word pattern wp_i of attribute A_i. S then sorts this list in ascending order starting from $id(S)$ to obtain list L and creates a message FwdResource$(id(S), id(p), p, L)$, where $id(p)$ is a unique metadata identifier assigned to p by S, and sends it to super-peer with identifier equal to $head(L)$. This forwarding is done as follows: message FwdResource is sent to a super-peer S', where $id(S')$ is the greatest identifier contained in the finger table of S, for which $id(S') \leq head(L)$ holds.

Upon reception of a message FwdResource by a super-peer S, $head(L)$ is checked. If $id(S) = head(L)$ then S removes $head(L)$ from list L and makes a copy of the message. The publication part of this message is then matched with the super-peer's local query database and subscribers are notified (the details of this are presented in Section 2.4). Finally, S forwards the message to super peer with identifier $head(L)$. If $id(S)$ is not in L, then it just forwards the message as described in the previous paragraph.

2.4 Notifying interested subscribers

Let us now examine how notifications about published resources are sent to interested subscribers. When a message FwdResource containing a publication p of a resource arrives at a super-peer S, the continuous queries matching p are found by utilising its local index structures and using the algorithm BestFitTrie presented in [11].

Once all the matching queries have been retrieved from the database, S creates a notification message of the form CQNotification$(id(C), l(r), L, T)$, where $l(r)$ is a link to the resource, L is a list of identifiers of the super-peers that are intended recipients of the notification message, and T is a list containing the query identifiers of the queries that matched p. List L is created as follows. S finds all super-peers that have at least one client with a query q satisfied by p. Then it sorts the list in ascending order starting from $id(S)$ and removes duplicate entries. The notification message is then forwarded according to the algorithm described in Section 2.3.

Upon arrival of a message CQNotification at a super-peer S, $head(L)$ is checked to find out whether S is an intended recipient of the message. If it is not, S just forwards the message to another super-peer using information from its finger table and the algorithm described in Section 2.3. If $head(L) = id(S)$, then S scans T to find the set U of query identifiers that belong to clients that have S as their access point, by utilising a hash table that associates query identifiers with client identifiers. For each distinct query identifier in set U, a message MatchingResource$(id(S), id(q), l(r))$ is created and forwarded to the appropriate client. Finally S removes $head(L)$ from L and U from T, and forwards message CQNotification according to the algorithm described in Section 2.3.

We are currently implementing the protocols described in Sections 2.2-2.4 to evaluate their performance and scalability.

3 Local Filtering Algorithms

In this section we present and evaluate BestFitTrie, a main memory algorithm that solves the filtering problem for *conjunctive queries* in $\mathcal{AWP}$. Algorithm BestFitTrie is run locally by each of the super-peers, and provides efficient matching of stored continuous queries against published resources. Because our work extends and improves previous algorithms of SIFT [12], we adopt terminology from SIFT in many cases. The material of this section appears in more detail in [11].

BestFitTrie uses two data structures to represent each published resource d: the *occurrence table* $OT(d)$ and the *distinct attribute list* $DAL(d)$. $OT(d)$ is a hash table that uses words as keys, and

is used for storing all the attributes of the resource in which a specific word appears, along with the positions that each word occupies in the attribute text. $DAL(d)$ is a linked list with one element for each distinct attribute of d. The element of $DAL(d)$ for attribute A points to another linked list, the *distinct word list* for A (denoted by $DWL(A)$) which contains all the distinct words that appear in $A(d)$.

To index queries BestFitTrie utilises an array, called the *attribute directory* (AD), that stores pointers to word directories. AD has one element for each distinct attribute in the query database. A *word directory* $WD(B_i)$ is a hash table that provides fast access to roots of *tries* in a *forest* that is used to organize *sets of words* – the set of words in wp_i (denoted by $words(wp_i)$) for each atomic formula $B_i \sqsupseteq wp_i$ in a query. The proximity formulas contained in each wp_i are stored in an array called the *proximity array* (PA). PA stores pointers to trie nodes (words) that are operands in proximity formulas along with the respective proximity intervals for each formula. Another hash table, called *equality table* (ET) indexes text values s_i that appear in atomic formulas of the form $A_i = s_i$.

When a new query q arrives, the index structures are populated as follows. For each attribute $A_i = s_i$, we hash text value s_i to obtain a slot in ET where we store the value A_i. For each attribute $B_j \sqsupseteq wp_j$, we compute $words(wp_j)$ and insert them in one of the tries with roots indexed by $WD(B_j)$. Finally, we visit PA and store pointers to trie nodes and proximity intervals for the proximity formulas contained in wp_j.

We now explain how each word directory $WD(B_j)$ and its forest of tries are organised. The idea is to store sets of words compactly by exploiting their *common elements*, to preserve memory space and to accelerate the filtering process.

Definition 1 *Let S be a set of sets of words and $s_1, s_2 \in S$ with $s_2 \subseteq s_1$. We say that s_2 is an* identifying subset *of s_1 with respect to S iff $s_2 = s_1$ or $\nexists\ r \in S$ such that $s_2 \subseteq r$.*

The sets of identifying subsets of two sets of words s_1 and s_2 with respect to a set S is the same if and only if s_1 is identical to s_2.

The sets of words $words(wp_j)$ are organised in the word directory $WD(B_j)$ as follows. Let S be the set of sets of words currently in $WD(B_j)$. When a new set of words s arrives, BestFitTrie selects an identifying subset t of s with respect to S and uses it to organise s in $WD(B_j)$. The algorithm for choosing t depends on the current organization of the word directory and will be given below.

Throughout its existence, each trie T of $WD(B_j)$ has the following properties. The nodes of T store sets of words and other data items related to these sets. Let $sets\text{-}of\text{-}words(T)$ denote the set of all sets of words stored by the nodes of T. A node of T stores more than one set of words iff these sets are identical. The root of T (at depth 0) stores sets of words with an identifying subset of cardinality one. In general, a node n of T at depth i stores sets of words with an identifying subset of cardinality $i+1$. A node n of T at depth i storing sets of words equal to s is implemented as a structure consisting of the following fields:

- $Word(n)$: the $i+1$-th word w_i of identifying subset $\{w_0, \ldots, w_{i-1}, w_i\}$ of s where $w_0, \ldots, w_{i-1}$ are the words of nodes appearing earlier on the path from the root to node n.
- $Query(n)$: a linked list containing the identifier of query q that contained word pattern wp for which $\{w_0, \ldots, w_i\}$ is the identifying subset of $sets\text{-}of\text{-}words(T)$.
- $Remainder(n)$: if node n is a leaf, this field is a linked list containing the words of s that are not included in $\{w_0, \ldots, w_i\}$. If n is not a leaf, this field is empty.
- $Children(n)$: a linked list of pairs (w_{i+1}, ptr), where w_{i+1} is a word such that $\{w_0, \ldots, w_i, w_{i+1}\}$ is an identifying subset for the sets of words stored at a child of w_i and ptr is a pointer to the node containing the word w_{i+1}.

The sets of words stored at node n of T are equal to $\{w_0, \ldots, w_n\} \cup Remainder(n)$, where $w_0, \ldots, w_n$ are the words on the path from the root of T to n. An identifying subset of these sets of words is $\{w_0, \ldots, w_n\}$. The purpose of $Remainder(n)$ is to allow for the delayed creation of nodes in trie. This delayed creation lets us choose which word from $Remainder(n)$ will become the child of current node n depending on the sets of words that will arrive later on.

The algorithm for inserting a new set of words s in a word directory is as follows. The first set of words to arrive will create a trie with the first word as the root and the rest stored as the remainder. The second set of words will consider being stored at the existing trie or create a trie of its own.

In general, to insert a new set of words s, BestFitTrie iterates through the words in s and utilises the hash table implementation of the word directory to find all *candidate tries* for storing s: the tries with root a word of s. To store sets as compactly as possible, BestFitTrie then looks for a trie node n such that the set of words $(\{w_0, \ldots, w_n\} \cup Remainder(n)) \cap s$, where $\{w_0, \ldots, w_n\}$ is the set of words on the path from the root to n, has maximum cardinality. There may be more than one node that satisfies this requirements and such nodes might belong to different tries. Thus BestFitTrie performs a depth-first search down to depth $|s| - 1$ in *all* candidate tries in order to decide the optimal node n. The path from the root to n is then extended with new nodes containing the words in $\tau = (s \setminus \{w_0, \ldots, w_n\}) \cap Remainder(n)$. If $s \subseteq \{w_0, \ldots, w_n\} \cup Remainder(n)$, then the last of these nodes l becomes a new leaf in the trie with $Query(l) = Query(n) \cup \{q\}$ (q is the new query from which s was extracted) and $Remainder(l) = Remainder(n) \setminus \tau$. Otherwise, the last of these nodes l points to two child nodes l_1 and l_2. Node l_1 will have $Word(l_1) = u$, where $u \in Remainder(n) \setminus \tau$, $Query(l_1) = Query(n)$ and $Remainder(l_1) = Remainder(n) \setminus (\tau \cup \{u\})$. Similarly node l_2 will have $Word(l_2) = v$, where $v \in s \setminus (\{w_0, \ldots, w_n\} \cup \tau)$, $Query(l_2) = q$ and $Remainder(l_2) = s \setminus (\{w_0, \ldots, w_n\} \cup \tau \cup \{u\})$. The complexity of inserting a set of words in a word directory is *linear* in the size of the word directory but *exponential* in the size of the inserted set. This exponential dependency is not a problem in practice because we expect *queries to be small* and the crucial parameter to be the size of the query database.

The filtering procedure utilises two arrays named $Total$ and $Count$. $Total$ has one element for each query in the database and stores the number of atomic formulas contained in that query. Array $Count$ is used for counting how many of the atomic formulas of a query match the corresponding attributes of a resource. Each element of array $Count$ is set to zero at the beginning of the filtering algorithm. If at algorithm termination, a query's entry in array $Total$ equals its entry in $Count$, then the query matches the published resource, since all of its atomic formulas match the corresponding resource attributes.

When a resource d is published, BestFitTrie hashes the text value $C(d)$ contained in each resource attribute C and probes the ET to find matching atomic formulas with equality. Then for each attribute C in $DAL(d)$ and for each word w in $DWL(C)$, the trie of $WD(C)$ with root w is traversed in a breadth-first manner. Only subtrees having as root a word contained in $C(d)$ are examined, and hash table $OT(d)$ is used to identify them quickly. At each node n of the trie, the list $Query(n)$ gives implicitly all atomic formulas $C \sqsupseteq wp$ that can potentially match $C(d)$ if the proximity formulas in wp are also satisfied. This is repeated for all the words in $DWL(C)$, to identify all the qualifying atomic formulas for attribute C. Then the proximity formulas for each qualifying query are examined using the polynomial time algorithm *prox* from [8]. For each atomic formula satisfied by $C(d)$, the corresponding query element in array $Count$ is increased by one. At the end of the filtering algorithm the equal entries in arrays $Total$ and $Count$ give us the queries satisfied by d.

To evaluate the performance of BestFitTrie we have also implemented algorithms BF, SWIN and PrefixTrie. BF (Brute Force) has no indexing strategy and scans the query database sequentially to determine matching queries. SWIN (Single Word INdex) utilises a two-level index for accessing queries in an efficient way. PrefixTrie is an extension of the algorithm Tree of [12] appropriately modified to cope with attributes and proximity information. Tree was originally proposed for storing *conjunctions of keywords* in secondary storage in the context of the SDI system SIFT. Following Tree, PrefixTrie uses *sequences* of words sorted in lexicographic order for capturing the words appearing in the word patterns of atomic formulas (instead of sets used by BestFitTrie). A trie is then used to store sequences compactly by exploiting *common prefixes* [12].

Algorithm BestFitTrie constitutes an improvement over PrefixTrie. Because PrefixTrie examines only the prefixes of sequences of words in lexicographic order to identify common parts, it misses many opportunities for clustering. BestFitTrie keeps the main idea behind PrefixTrie but searches exhaustively the current word directory to discover the best place to introduce a new set of words. This allows BestFitTrie to achieve better clustering as PrefixTrie introduces redundant nodes that are the result of using a lexicographic order to identify common parts. This node redundancy can be the cause of deceleration of the filtering process as we will show in the next section. The only way to improve beyond BestFitTrie would be to consider *re-organizing* the word directory every time a new set of words arrives, or periodically. We have not

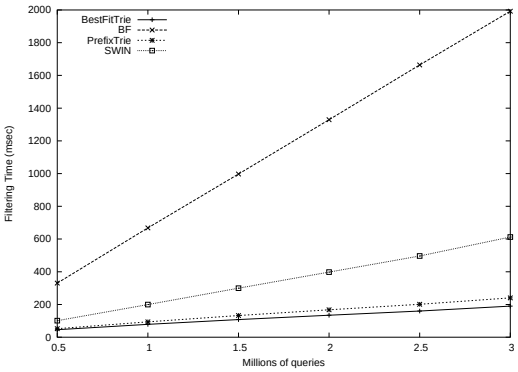

Figure 2: Effect of the query database size in filtering time

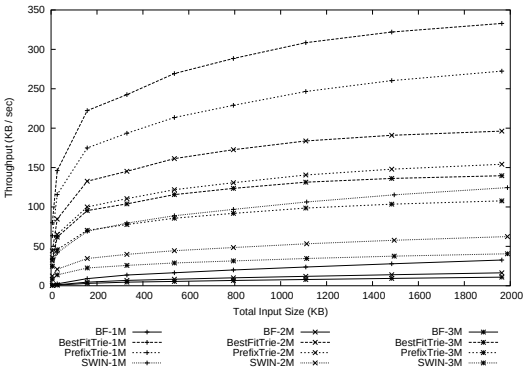

Figure 3: Throughput achieved as a function of total input size

explored this approach in any depth.

3.1 Experimental Evaluation

We evaluated the algorithms presented above experimentally using a set of documents downloaded from ResearchIndex[2] and originally compiled in [5]. 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, we developed a methodology for creating user queries using *words* and *technical terms* (phrases) extracted automatically from the ResearchIndex documents using the C-value/NC-value approach of [5].

All the algorithms were implemented in C/C++, and the experiments were run on a PC, with a Pentium III 1.7GHz processor, with 1GB RAM, running Linux. The results of each experiment are averaged over 10 runs to eliminate any fluctuations in the time measurements.

The first experiment targeted the performance of algorithms under different query database sizes. In this experiment, we randomly selected one hundred documents from the NN corpus and used them as incoming resources in query databases of different sizes. The size and the matching percentage for each resource used was different but the average resource size was 6869 words, whereas on average 1% of the queries stored matched the incoming resources.

As we can see in Figure 2, the time taken by each algorithm grows linearly with the size of the query database. However SWIN, PrefixTrie and BestFitTrie are less sensitive than Brute Force to changes in the query database size. The trie-based algorithms outperform SWIN mainly due to the clustering technique that allows the exclusion of more non-matching atomic queries. We can also observe that the better exploitation of the commonalities between queries improves the performance of BestFitTrie over PrefixTrie, resulting in a significant speedup in filtering time for *large query databases*. Figure 3 shows that BestFitTrie gives the best filtering performance by processing a load of about 150KB (about 9 ResearchIndex papers) per second for a query database of 3 million queries.

In terms of space requirements BF needs about 15% less space than the trie-based algorithms, while the rate of increase for the two trie-based algorithms is similar to that of BF, requiring a fixed amount of extra space each time. Thus it is clear that BestFitTrie speeds up the filtering process with a small extra storage cost, and proves faster than the rest of the algorithms, managing to filter user queries about 10 times faster than the sequential scan method. Finally, the query insertion rate that the two trie-based algorithms can support is about 40 queries/second for a database containing 2.5 million queries.

We have also evaluated the performance of the algorithms under two other parameters: *resource size* and *percentage of queries matching a published resource*. Finally we have developed various heuristics for ordering words in the tries maintained by PrefixTrie and BestFitTrie when *word frequency* information (or *word ranking*) is available as it is common in IR research [2]. The details of these experiments are omitted due to space considerations.

[2] `http://www.researchindex.com`

4 Work in Progress

Performance evaluation in the distributed case. To evaluate the performance and scalability of DHTrie we are currently implementing the algorithms of Section 2. We plan to evaluate DHTrie by considering its behaviour (mainly expressed in terms of message load between super-peers) under various parameters (query and resource size, arrival rates of queries and resources, number of super-peers, etc.), and also by comparing it to other alternatives such as flooding and schema-based routing of queries and resources. The results of the evaluation will be presented at the workshop.

Load balancing. A key problem that arises when trying to partition the query space among the different super-peers in our overlay network is *load balancing*. The idea here is to avoid having overloaded peers i.e., peers having to handle a great number of posted queries (this is what [1] calls *storage load balancing*; although the paper [1] is not in a pub/sub setting, the concept is the same).

In addition, we would like to have a way to deal with the load balancing problem posed to super-peers that are responsible for pairs (A, w), where word w appears frequently in text values involving A. We expect the frequency of occurrence of words appearing in a query within an atomic formula with attribute A to follow a non-uniform distribution (e.g., a skewed distribution like the Zipf distribution [13]). We do not know of any study that has shown this by examining collections of user queries; however, such an assumption seems intuitive especially in the light of similar distributions of words in text collections [2]. As an example, in a digital library application we would expect distinguished author names to appear frequently in queries with the AUTHOR attribute, or popular topics to appear frequently in queries with the TITLE attribute. Thus in our case, uniformity of data items (i.e., queries) as traditionally assumed by DHTs is not applicable.

We are currently working on addressing the above load balancing problems by utilizing ideas from the algorithm LCWTrie described in detail in [11, 7] where queries are indexed under infrequent words; we also use a form of controlled replication to deal with overloading due to notification processing. It would be interesting to compare this approach to what is advocated in [1].

Word frequency computation in a distributed setting. Computing the frequency of occurrence of words in a distributed setting is a crucial problem if one wants to support vector space queries or to provide for load balancing among peers as showed above. There are mainly two approaches to the word frequency computation in a distributed setting; (a) a global ranking scheme that assumes a central authority that maintains the frequency information or a message-intensive update mechanism that notifies every peer in the network about changes in frequency information or (b) a local ranking scheme that computes word frequencies of peer p_i based solely on frequencies of words in documents that are published at p_i. It is clear that the first approach affects the scalability and efficiency of the system while the second approach can be misleading due to peer specialisation.

In related work we present a distributed word ranking algorithm that is a hybrid form of the two approaches described earlier. It provides an algorithm that is based on local information, but also tries to combine this information with the global "truth" through an updating and estimation mechanism.

Reducing network traffic. We can reduce network traffic by *compressing* publications. In the full version of the paper we describe a gap compression technique that allows the matching of a compressed publication against a database of user queries using algorithm BestFitTrie.

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, Swiss Federal Institute of Technology Lausanne (EPFL), 2004.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[3] A. Carzaniga, D.-S. Rosenblum, and A.L Wolf. Design and evaluation of a wide-area event notification service. *ACM TOCS*, 19(3):332–383, August 2001.

[4] P. A. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl. Publish/Subscribe for RDF-based P2P Networks. In *Proceedings of the 1st European Semantic Web Symposium*, May 2004.

[5] L. Dong. Automatic term extraction and similarity assessment in a domain specific document corpus. Master's thesis, Dept. of Computer Science, Dalhousie University, Halifax, Canada, 2002.

[6] S. Idreos, M. Koubarakis, and C. Tryfonopoulos. P2P-DIET: Ad-hoc and Continuous Queries

in Super-Peer Networks. In *Proceedings of the IX International Conference on Extending Database Technology (EDBT04)*, pages 851–853, Heraklion, Crete, Greece, 14–18 March 2004.

[7] S. Idreos, C. Tryfonopoulos, M. Koubarakis, and Y. Drougas. Query Processing in Super-Peer Networks with Languages Based on Information Retrieval: the P2P-DIET Approach. In *Proceedings of P2P & DB 2004*, 2004.

[8] M. Koubarakis, C. Tryfonopoulos, P. Raftopoulou, and T. Koutris. Data models and languages for agent-based textual information dissemination. In *Proceedings of CIA 2002*, volume 2446 of *LNCS*, pages 179–193, September 2002.

[9] M. Koubarakis and T. Koutris and P. Raftopoulou and C. Tryfonopoulos. Information Alert in Distributed Digital Libraries: The Models, Languages and Architecture of DIAS. In *Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries (ECDL 2002)*, volume 2458 of *Lecture Notes in Computer Science*, pages 527–542, September 2002.

[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 *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[11] C. Tryfonopoulos, M. Koubarakis, and Y. Drougas. Filtering Algorithms for Information Retrieval Models with Named Attributes and Proximity Operators. In *Proceedings of the 27th Annual ACM SIGIR Conference*, Sheffield, United Kingdom, July 25-July 29 2004. Forthcoming.

[12] 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.

[13] G. K. Zipf. *Human Behaviour and Principle of Least Effort*. Addison Wesley, Cambridge, Massachusetts, 1949.