Distributed Pattern Matching: A Key to Flexible and Efficient P2P Search

Reaz Ahmed and Raouf Boutaba
School of Computer Science, University of Waterloo, Ontario, Canada N2L 3G1
Email: {r5ahmed, rboutaba}@uwaterloo.ca, Telephone: (519) 888–4820, Fax: (519) 885–1208

Abstract— **Flexibility and efficiency are the prime requirements for any P2P search mechanism. Existing P2P systems do not seem to provide satisfactory solution for achieving these two conflicting goals. Unstructured search protocols (as adopted in Gnutella and FastTrack) provide search flexibility but exhibit poor performance characteristics. Structured search techniques (mostly Distributed Hash Table (DHT)-based), on the other hand, can efficiently route queries but support exact-match semantic only.**

In this paper we have defined Distributed Pattern Matching (DPM) problem and have presented a novel P2P architecture, named Distributed Pattern Matching System (DPMS), as a solution. Possible application areas of DPM include P2P search, service discovery and P2P databases. In DPMS, advertised patterns are replicated and aggregated by the peers, organized in a lattice-like hierarchy. Replication improves availability and resilience to peer failure, and aggregation reduces storage overhead. An advertised pattern can be discovered using any subset of its 1-bits. Search complexity in DPMS is *logarithmic* to the total number of peers in the system. Advertisement overhead and guarantee on search completeness is comparable to that of DHT-based systems. We have presented mathematical analysis and simulation results to demonstrate the effectiveness of DPMS.

I. Introduction

The generic problem of pattern matching and its variants have extensively been studied in Computer Science literature. In this paper we have defined Distributed Pattern Matching (DPM) as a variant of the generic pattern matching problem with two additional constraints. First, we are interested in Bloom-filter [5] based pattern matching (*i.e.*, subset matching), and second, we assume that the patterns are scattered among the peers of a P2P overlay network (see Fig. 1). Given a search pattern Q, the goal is to find the peer(s) containing a pattern (say P) matching Q. P matches Q if $P \wedge Q = Q$; *i.e.*, the 1-bits of Q is a subset of the 1-bits of P. We assume a pattern to be a bloom filter (a couple of hundred bits in length) constructed by hashing the properties of a shared object (such as a file or a service).

Problems that can be mapped to DPM include a) partial- and multi-keyword search for content sharing P2P systems, b) partial service description matching for service discovery systems, c) data record pre-scanning for distributed P2P database systems, d) molecular fingerprint matching in some envisioned distributed environment, *etc*.

In this paper we have presented a novel P2P system, DPMS (Distributed Pattern Matching System), for efficiently solving the DPM problem. We have also demonstrated the application of DPMS in solving partial keyword matching problem. We have provided mathematical analysis and simulation results for establishing the effectiveness of the proposed architecture.

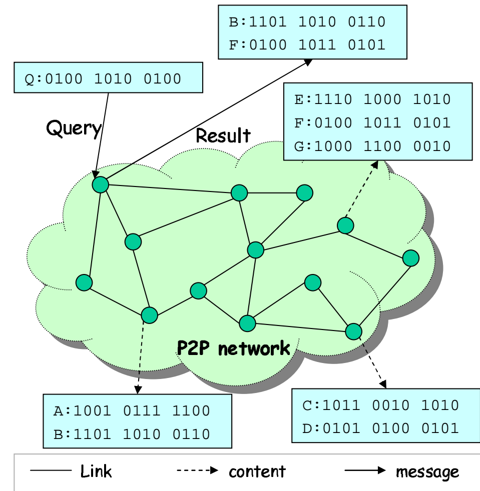


Fig. 1. The Distributed Pattern Matching (DPM) problem

DPMS achieves many desirable properties of both the unstructured and structured P2P systems. Like unstructured systems DPMS supports partial keyword matching, utilizes the heterogeneity in peer capabilities, and does not place any hard restriction on index/document placement. Like structured systems, on the other hand, DPMS attains logarithmic bound on search complexity and offers guarantee on search completeness and discovery of rare items. Advertisement traffic in DPMS is comparable to that of DHT-based structured P2P systems.

To our knowledge Distributed Pattern Matching (DPM) problem has never been addressed by any research activity in P2P context. [1]The index distribution architecture of DPMS is unique and has been designed specifically to solve the DPM problem. The novel aggregation scheme, proposed in this paper, can effectively reduce storage overhead at the indexing peers without incurring a significant decrease in query routing performance. However, the use of bloom filter for representing indices is not new. Many network applications use bloom filters. A comprehensive list of such applications can be found in [6].

The rest of this paper is organized as follows. Section II highlights and compares the approaches related to DPMS. The architecture and operation of DPMS are presented in section III. Mathematical analysis of search complexity in DPMS is provided in section IV. Experimental results and

[1]This work is an extension of our previous publication [3].

concluding remarks are presented in section V and section VI, respectively.

II. Related Works

Existing solutions for pattern matching [4], [10] in centralized environment, hold linear relationship with the number of advertised patterns (or *text* according to pattern-matching literature) to be matched. This implies flooding for an equivalent solution to the DPM problem.

From architectural point of view, Secure Service Discovery Service (SSDS) [11] is the closest match to DPMS. Like DPMS SSDS uses Bloom filter and aggregation. However, index distribution in SSDS is through a tree-like hierarchy of indexing nodes, in contrast to the lattice-like hierarchy used by DPMS. SSDS does not use any replication in the indexing hierarchy. Higher level nodes in SSDS index tree handle higher volume of query/advertisment traffic and the system is more sensitive to the failure of these nodes. Another major drawback of SSDS, compared to DPMS, lies in its aggregation mechanism. SSDS uses bitwise logical-OR for index aggregation. The aggregation scheme adopted in DPMS (explained in section III-C) retains unchanged bits from constituent patterns and provides more useful information during query routing.

Unstructured systems ([2],[1]) identify objects by keywords. Advertisements and queries are in terms of the keywords associated with the shared objects. Structured systems, on the other hand, identify objects by keys generated by applying one-way hash function on keywords associated with an object. Key-based query routing is much efficient than keyword-based unstructured query routing. The downside of key-based query routing is the lack of support for partial-matching semantics. Unstructured systems, utilizing blind search methods (like flooding[2] and random-walks [15]), can easily be modified to facilitate partial-matching of queries, and in general to solve DPM problem. Due to the lack of proper routing information, the generated query routing traffic would be very high. Besides, there would be no guarantee on search completeness.

Many research activities are aimed at improving the routing performance of unstructured P2P systems. Different routing hints are used in different approaches. In [7] routing is biased by peer capacity; queries are routed to peers of higher capacity with higher probability. In [23] and [21] peers learn from the results of previous routing decisions and bias future query routing based on this knowledge. In [9] peers are organized based on common interest. Restricted flooding is performed in different interest groups. Many research papers ([7], [23], [13], *etc.*) propose storing index information from peers within a radius of 2 or 3 hops on the overlay network. All of these techniques reduce volume of query traffic to some extent, but do not provide guarantee on search completeness.

Bloom filter is used by many unstructured P2P systems for improving query routing performance. In [13] each peer stores Bloom filters from peers one or two hops away. Experimental results presented in [13] show that logical OR-based aggregation of Bloom filters is not suitable for aggregating information from peers more than one hop away. In [18] each peer store a list of Bloom filters per neighbor. The ith Bloom filter in the list for neighbor, say M, summarizes the documents that are $i-1$ hops away via M. A query is forwarded to the neighbor with a matching Bloom filter at the smallest hop-distance. This approach aims to find the closest replica of a document with a high probability.

Schmidt et. al. [19] have presented an approach, named Squid, for supporting partial keyword matching in DHT-based structured P2P networks. They have adopted space-filling-curves to map similar keywords to numerically close keys. Squid supports partial prefix matching (*e.g.*, queries like compu* or net*) and multi keyword queries. Squid does not support true wildcard matching for queries like **net**. Another extension to the DHT-technique for solving partial-keyword matching has been proposed in [12]. A keyword can be fragmented into η-grams, and each η-gram can be hashed and stored at the responsible peer. This approach can solve partial keyword matching problem. However, solving the generic DPM problem with this approach is not feasible.

In general DHT-techniques ([20], [17], [16], [24] *etc.*) are not suitable for solving partial keyword matching problem (and DPM problem) for two reasons. Firstly, DHT-techniques require to partition the key-space into non-overlapping regions and to assign each region to a peer bearing an ID from that region. But from pattern matching perspective it is quite difficult to partition even one dimensional pattern (or key) space into non-overlapping clusters, while preserving closeness of patters in hamming distance. Secondly, DHT-techniques cannot handle *common keywords problem* [14] well. Popular η-grams like ”tion” or ”ing” can incur heavy load on the peers responsible for these η-grams, resulting into unequal distribution of load among the participating peers.

III. Distributed Pattern Matching System (DPMS)

This section presents details on DPMS architecture. In this section we will use the terms *pattern* and *index* interchangeably, as patterns are used as indices for query routing.

A. Overview

In DPMS a peer can act as a *leaf peer* or *indexing peer*. A leaf peer resides at the bottom level of the indexing hierarchy and advertises its indices (created from the objects it is willing to share) to other peers in the system. An indexing peer, on the other hand, stores indices from other peers (leaf peers or indexing peers). A peer can join different levels of the indexing hierarchy and can simultaneously act in both the roles. Indexing peers get arranged into a lattice-like hierarchy (see Fig. 2) and disseminate index information using repeated aggregation and replication.

DPMS uses replication trees (see Fig. 2a) for disseminating patterns from leaf peers to a large number of indexing peers. However, such a replication strategy would generate a large volume of advertisement traffic. To overcome this shortcoming, DPMS combines replication with lossy-aggregation. As shown in Fig. 2b, advertisements from different peers are aggregated and propagated to peers in the next level along the aggregation tree.

The structure of the indexing hierarchy and the amount of replication are controlled by two system-wide parameters, namely replication factor R and branching factor B. Patterns advertised by a leaf peer are propagated to R^l indexing peers at level l. On the other hand, an indexing peer at level l contains patterns from B^l leaf peers. Due to repeated (lossy) aggregation, information content of the aggregates reduces as we go up along the indexing hierarchy.

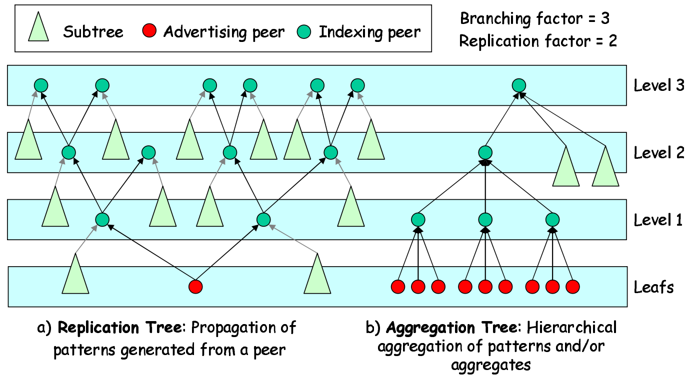


Fig. 2. DPMS overview

The indexing hierarchy has three-fold impact on system performance. Firstly, the indexing hierarchy evenly distributes index information (and queries) in the highest level indexing peers. This helps in load balancing the system and improves fault tolerance. Secondly, peers can route queries towards a target leaf peer without having any global knowledge of the overlay topology. Finally, the indexing hierarchy helps in minimizing query forwarding traffic. While forwarding a query from a root peer to multiple leaf peers in the same aggregation tree, shared path from the root peer to the common ancestor of the target leaf peers is utilized.

B. Index/pattern Construction

DPMS uses Bloom filters [5] as indices. Bloom filter is a space-efficient data structure used for set membership tests. However, this space-efficiency comes at the expense of a small possibility of false positive in the membership check operation.

The algorithm for Bloom filter construction is simple. A Bloom filter is represented as an m-bit array. k different hash functions are also required to be defined. Each of these hash functions should return values within the range of $\{0, \ldots, m-1\}$. In an empty Bloom filter all of the m-bits are set to 0. To insert an element (a string or keyword), it is hashed with the k hash functions and corresponding k array positions are set to 1. To test set membership for an element, it is hashed with the same k hash functions to get k array positions. If all of these k-bits are set (*i.e.*, 1) then with high probability the element is a member of the set represented by the Bloom filter, otherwise it is not. False positive probability for a membership test is calculated as $\epsilon = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k$, where n is the number of elements inserted in the Bloom filter. ϵ is minimized when $k = \ln 2 \cdot (m/n)$. For example with $m/n = 8$ and $k = 5$, $\epsilon \approx 0.02$.

A document in a traditional file-sharing P2P system is associated with a set of keywords. In DPMS all the keywords associated with a document are encoded in a single bloom-filter. To facilitate wildcard matching, each keyword is first fragmented into η-grams (usually trigrams). These η-grams are then inserted into the Bloom filter representing the document. Query keywords are also fragmented into η-grams (see Fig. 3) and encoded into a Bloom filter. The 1-bits on a query should be a subset of the set of 1-bits of any pattern that it should match against.

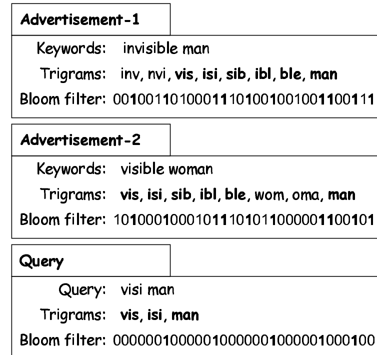


Fig. 3. Index construction example; query *"∗visi∗man∗"* matches advertised names *"invisible man"* and *"visible woman"*

For a P2P service discovery system indices can be obtained in a similar fashion, using attribute-value pairs instead of keywords. Molecular fingerprint can be used as index for some envisioned distributed system storing molecular structure information.

C. Aggregates

An aggregate is obtained by combining two (or more) different patterns (or aggregates). DPMS index distribution and query routing architecture is independent of the underlying aggregation scheme. It is possible to plug-in different aggregation schemes with DPMS. However, all of the peers in a system must use the same aggregation mechanism. An aggregation mechanism should have the following properties to be compatible with the DPMS indexing hierarchy:

- The aggregation scheme should compress index information obtained from child peers. Lossy compression is allowed. Parameter control over the level of aggregation if preffered.
- The aggregated form should retain original pattern information (to some extent), making it possible to perform pattern matching on the aggregates.
- Repeated aggregation should be possible, *i.e.*, it should be possible to perform aggregation on aggregates without violating the pervious requirements.

A trivial way of aggregation is to OR the bits in the patterns to be aggregated (as adopted in [13] and [11]). But the information loss in this aggregation scheme is very high. Moreover, while matching a query with an aggregate, we cannot say that some subset of the 1-bits in the aggregate was present in one single constituent pattern.

Considering the requirements and the problems with OR-based aggregation, we suggest a don't care based aggregation

scheme; don't cares (represented by X) are used at the positions where the constituent patterns (or aggregates) differ. We will use $\otimes$ to denote the aggregation operation. If $Q = A \otimes B$, then the ith bit of Q is obtained as,

$$q_i = \begin{cases} a_i & \text{if } a_i = b_i \\ X & \text{otherwise, } i.e., a_i \neq b_i \text{ or } a_i = X \text{ or } b_i = X. \end{cases}$$

This type of aggregates retain parts from the constituent patterns or aggregates. A 1-bit (or 0-bit) in such an aggregate indicate that all of the patterns contributing to this aggregate had 1 (or 0) at the corresponding position. However incorporating this extra information (*i.e.*, X's) incur some space overhead, which can be minimized by compressing the aggregates using huffman coding or run length encoding during transmission through the network.

D. Aggregation Process

An indexing peer acts as a multiplexer in the indexing hierarchy. It gathers *in-lists* (lists of patterns or aggregates from the B child peers), aggregates them to another list (referred to as *out-list*) of aggregates, and sends this list to each of its parents.

Construction of out-list is not trivial. We want the aggregates in the out-list to have a minimum number of X-bits. This ensures minimum information loss. The problem of obtaining an out-list containing a minimum number of X-bits is NP-complete. Instead we use a heuristic approach to obtain approximate solution. The formula for measuring the similarity of two patterns/aggregates, say Q and R, is given in (1). Table I presents the significance of different terms in (1).

We have used $Q^t = \{q_i^t | q_i^t \in \{0,1\} \wedge t \in \{0,1,X\} \wedge q_i = t \Rightarrow q_i^t = 1\}$ to define a mask on Q [2], and $|Q^t|$ to denote the cardinality of Q^t. Hence, $(|Q^1| + |Q^0| + |Q^X|)$ is the length of Q.

$$h(Q,R) = \frac{E + \alpha \times F + \beta \times G - \gamma \times H}{|Q^1| + |Q^0| + |Q^X|} \quad (1)$$

In this equation E and F define the number of positions in the aggregate that will remain the same as that of Q (or R). While G and H give a measure of relative increase of X-bits in the resulting aggregate. Coefficients α, β and γ depend on the nature of the patterns, which may require system specific tuning. For the simulations presented in this paper, we have used $\alpha = 0.33$, $\beta = 0$ and $\gamma = 0.33$. The values of these coefficients have to increased if the advertised patterns exhibit negative correlation.

The aggregation algorithm (Algorithm. 1) takes the following three parameters and generates the out-list.

- *In-list* $(Pattern[])$ is an array of patterns or aggregates constructed from the B in-lists received from the B children.
- *Minimum non-X bits (O)* is the minimum number of original (*i.e.*, non-X) bits an aggregate *must* retain after aggregation.
- *Target aggregation ratio (A)* is the ratio of the number of aggregates in the out-list to the number of patterns/aggregates in the in-list. The aim is to achieve an aggregation ratio of A without violating the constraint imposed by O.

[2] For example, if Q = `0X1X 101X`, then we can compute Q^1 = `0010 1010`, Q^0 = `1000 0100` and Q^X = `0101 0001`

TABLE I
SIGNIFICANCE OF DIFFERENT COMPONENTS IN (1)

q_i	r_i	coeff.	term	computed as
0	0	1	E	$\|Q^0 \wedge R^0\|+$
1	1			$\|Q^1 \wedge R^1\|$
X	X	α	F	$\|Q^X \wedge R^X\|$
0	X			
1	X	β	G	$\|Q^X \oplus R^X\|$
X	0			
X	1			
0	1	$-\gamma$	H	$\|Q^0 \wedge R^1\|+$
1	0			$\|Q^1 \wedge R^0\|$

Algorithm 1 Aggregate a list of Patterns

```
1:  Input: inList : Pattern[], O : Integer, A : Float
2:  Output: outList : Pattern[]
3:  Global: h(P,Q) : see (1)
4:          W : pattern width
5:  outList ← inList
6:  while |outList| > A × |inList| do
7:      find P_r ∈ outList and Q_r ∈ outList such that
            (P_r ≠ Q_r)∧
            (|(P_r ⊗ Q_r)^X| < W − O)∧
            (h(P_r, Q_r) ≥ h(P,Q) ∀P,Q ∈ outList)
8:      if no such P_r and Q_r exists then
9:          break {failed to achieve target aggregation ratio}
10:     end if
11:     P_n ← P_r ⊗ Q_r
12:     outList ← {outList − {P_r, Q_r}} ∪ {P_n}
13: end while
14: return outList
```

E. Index Distribution

Indexing peers at level l arrange into R^l groups, numbered from 0 to $(R^l - 1)$ (see Fig. 4). In the ideal case, all the indexing peers in a single group (at any level) collectively cover all the leaf peers in the system.

A peer at level l and group g $(0 \leq g < R^l)$ is responsible for transmitting its aggregated information to R parents at level $(l+1)$. Each parent belongs to a different group in range $[g \times R, (g+1) \times R)$, respectively.

Peers at level l and group g organize into subgroups (referred to as siblings) of size B to forward their aggregated information to the same set of parents. Thus each group in range $[g \times R, (g+1) \times R)$ at level $(l+1)$ will contain a peer replicating the same index information. This provides redundant paths for query routing and increases tolerance to peer failure.

F. Query Routing

A query can be initiated by any peer in the system. The query life-cycle can be divided into three phases: ascending phase, blind search phase and descending phase.

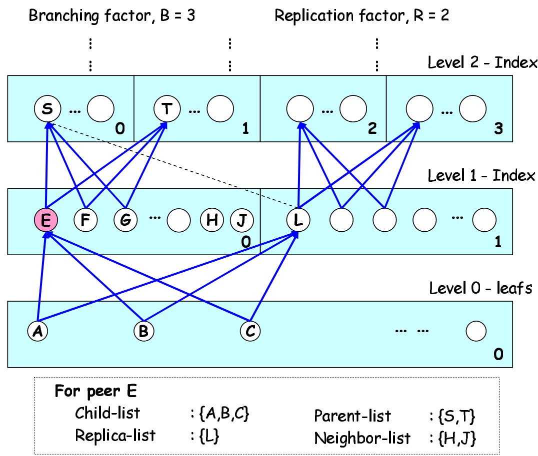


Fig. 4. Index distribution architecture. All the peers interacting with peer E are labelled. Group number is printed at the bottom right corner of each box.

During the *ascending phase*, an initiating (or intermediate) peer checks its local information for the existence of a match. If a match is found then the query is forwarded to the matching child, otherwise it is forwarded to any of its parents. This precess recurs until the query hits a peer with a match, or reaches a highest level peer.

The *Blind search phase* is executed by a highest level peer, say Z, upon receiving a query (from a child) that does not match any aggregate in its aggregate-lists. Z floods the query to all other peers in its group. If no peer in a group at the highest level contains a match then the query was for a non-existent pattern, and so the search fails.

A query enters into the *descending phase* when it hits a peer containing a matching aggregate. The query is then forwarded to the child peer advertising the matching aggregate. This process recurs until the query reaches a leaf peer. Two types of exceptions may occur. Firstly, a false match may occur and the search branch terminates. Secondly, a peer may have multiple children matching the query and multiple search branches can be initiated. Priority and the order in which search branches are initiated is guided by predefined policy and application-specific requirements. A possible scale for measuring the quality of a match is $\frac{|P^1 \wedge Q^1| + |P^0 \wedge Q^0|}{|P^0| + |P^1| + |P^X|}$, *i.e.*, the proportion of exact-match of a query Q with an aggregate P.

We prefer iterative version of query forwarding over recursive version for two reasons. Firstly, the number of simultaneous search branches can easily be controlled. Secondly, search termination criteria (*e.g.*, maximum probes, results per query *etc.*) can be handled more flexibly.

A useful feature of DPMS is context-sensitive routing, which stems from the existence of the ascending phase. Peers in similar context (*e.g.*, network proximity, common interest *etc.*) can advertise their aggregates along the same aggregation tree. This will automatically influence the query routing mechanism to select the nearest leaf peer containing a matching pattern.

G. Topology Maintenance

In the DPMS index distribution hierarchy, peers interact with each other in different roles, *e.g.*, parent, child, neighbor *etc*. An indexing peer, say E, at level l and group g, maintains four separate lists for this purpose (see Fig. 4).

1) *Replica-list* contains the list of peers in the adjacent groups that have common children as that of E. This list contains $(R-1)$ peers, one from each group in range $[\lfloor g/R \rfloor \times R, (\lfloor g/R \rfloor + 1) \times R)$, excluding g.
2) *Parent-list* This is the replica list obtained from one of E's parents. E uses this list to forward its aggregate information (out-list) to all of its parents along a replication tree.
3) *Child-list* contains the list of all children and the replica-list for each of them. A peer normally communicates with the child peers only. But in case of a failure of a child it can communicate with a replica of the failed child. This list contains B entries corresponding to the B children of E at level $(l-1)$ and group g/R.
4) *Neighbor-list* contains a fixed number of non-sibling peers that are in the same group (g) as peer E. This list is used for maintaining connectivity in a group, during join operation, and for flooding queries horizontally within a group (usually at the topmost level).

Out of these four lists a peer needs to keep track of three: child-list, parent-list and neighbor-list. The replica-list of a peer is the parent-list of any of its children. Peers use the Newscast protocol [22] for maintaining and updating these lists, *i.e.*, to detect peer failures and arrival of new peers. Flow of news packets is restricted to $2 \times R$ groups of peers. More specifically, the news-list of a peer at level l and group g will contain information about some peers from groups $[\lfloor g/R \rfloor \times R, (\lfloor g/R \rfloor + 1) \times R)$ at level l and groups $[g \times R, (g+1) \times R)$ at level $(l+1)$. That is, each peer sends news packets to its parents, neighbors and replicas, and receives news packets from its children, neighbors and replicas.

Unlike indexing peers, a leaf peer maintains only the neighbor-list and forwards this list to its parents. A leaf peer obtains its parent-list from one of its parents. It should be noted that leaf peers do not have any replica-list or child-list.

H. Arrival of Peers

To join as a leaf a peer, say C, has to find a level 1 indexing peer, say P, with an empty slot in its child-list. C joins the indexing hierarchy as a child of P. C constructs its parent-list using the replica-list of P, and starts advertising its patterns to all of its parents. If C fails to find a level 1 peer with an empty slot then it can either join in both level 1 and level 0, or select a level 1 peer with smaller number of children.

To join as an indexing peer, a peer, say E, has to go through the following steps:

- *Choose level and group*: E has to choose a level, say l, in the hierarchy. Selection of level can be based on the nodes capacity, uptime distribution *etc.* Peers with higher capacity (storage and bandwidth) and longer life-time are expected to join higher levels in the indexing hierarchy. Then peer E can choose a group g in random such that g is in $[0, R^l)$.
- *Construct child-list*: E has to contact a seed peer to get information about other peers in the system. E can crawl

the indexing hierarchy to reach a peer, say A, such that A is at level $(l-1)$ and in group $\lfloor g/R \rfloor$, and the parent-list of peer A contains less than R entries. E can join as a parent of A. E has to join the group in which A has no parents. E has to obtain and update the replica-list of other parents of A. E then obtains the child-list from a parent of A. A can have an empty parent-list during the initialization phase of the system, or after a failure of all of its parents. If A returns an empty parent-list, then E should look for other (up to B) peers, in the same group as that of A, with empty parent-list. If such peers exist then E should make them its children.

- *Construct parent-list*: To construct parent-list, E has to find a peer, say T, such that T is in level $(l+1)$ and in group $(g \times R)$, and T has an empty slot in its child-list. If such a peer (T) exists then E constructs its parent list using T and all the replicas of T. Otherwise, E will start with an empty parent-list, and will wait for more peers to join at level $(l+1)$.

I. Departure/Failure of Peers

In DPMS, peer departure and failure are handled in the same manner. The absence of an indexing peer, say E, will affect the peers in its parent-list, child-list and replica-list. Parents and children of E can still communicate through any of the replicas of E. So query routing is not hampered until all of the replicas of a peer fail.

Failure or departure of a leaf peer has greater impact on the system. All the index information along the replication tree, rooted at the failed leaf peer, has to be updated. During this period (from the point of failure to the update of all indexing peers in the replication tree) queries directed towards the failed leaf peer will be evaluated as false matches. This will decrease search performance to some extent.

To efficiently deal with frequent join and leave of a leaf peer, indexing peers should advertise their index information at constant intervals. Any advertisement from a child peer should be delayed until the end of the interval. The interval length can be used to tradeoff index update delay with network overhead due to frequent advertisements.

IV. Analysis

A. Query Routing Efficiency

In this section we will provide an analytical bound on the levels of indexing hierarchy that will allow query routing in $O(\log N)$ hops, where N is the total number of peers in the system. For this analysis we will use B to denote branching factor, R for replication factor, and n_l as the number of peers at level l. Leaf peers reside at level 0 and the height (or maximum level) of the indexing hierarchy is h. Assuming these definitions we can calculate the total number of peers at level l as:

$$n_l = n_0 \left(\frac{R}{B}\right)^l = n_0 \alpha^l \tag{2}$$

and the total number of peers in the system as:

$$N = \sum_{l=0}^{h} n_l = n_0 \frac{\alpha^{h+1} - 1}{\alpha - 1} \tag{3}$$

Hence the total number of leaf peers in the system,

$$n_0 = N \frac{\alpha - 1}{\alpha^{h+1} - 1} \tag{4}$$

Now, the number of groups at level l is R^l. So, the average number of peers in a group at level l, say m_l, is:

$$m_l = \frac{n_l}{R^l} \tag{5}$$

Combining (2), (3) and (5), and using $\alpha = \frac{R}{B}$ we obtain:

$$m_l = \frac{N(\alpha - 1)}{B^l(\alpha^{h+1} - 1)} \tag{6}$$

For efficient query routing we expect the number of peers in a group at level h to be $f \times \log N$, where $0.5 \leq f \leq 2$. Replacing $m_h = f \times \log N$ in (6) we obtain:

$$B^h(\alpha^{h+1} - 1) = \frac{N(\alpha - 1)}{f \log N} \tag{7}$$

For practical values of α and h we can approximate $(\alpha^{h+1} - 1)$ with $(\theta \times \alpha^h)$ for some constant θ. Replacing this value in (7) we can approximate h (for $R \neq B$ and $R > 1$) as:

$$h \approx \frac{1}{\log R} \times \log \frac{N \times (\alpha - 1)}{\theta f \log N} = O\left(\log \frac{N}{\log N}\right) \tag{8}$$

Thus we claim that if we can build and maintain an indexing hierarchy of height $O\left(\log \frac{N}{\log N}\right)$ then we will be able to solve the DPM problem in $O\left(\log N + \xi\kappa \log \frac{N}{\log N}\right)$ time. Here, $O(\log N)$ is the cost of flooding one of the R^h groups at level h, and $O(\xi\kappa \log \frac{N}{\log N})$ is the cost of reaching the κ matching leaf peers along the indexing hierarchy of height $O(\log \frac{N}{\log N})$. ξ accounts for the lossy aggregation scheme. For a system without any aggregation (*i.e.*, information loss) the value of ξ should equal one. Section IV-C presents an estimate of ξ.

B. False Match Probability

In this section we establish the effectiveness of don't care based aggregation scheme over OR based aggregation[3] as adopted in [11] and [13]. In don't care based aggregation scheme, we consider an aggregate D to be a match for a query Q if the set of 1-bits in Q is a subset of the 1-bits in D, assuming the don't care positions in D to be 1s. This assumption leads to the possibility of *false match*, where an aggregate can match a query, although none of the constituent patterns is a match for the query. For measuring the false match probability we will assume that the pattern's (advertised by the leaf peers) are Bloom-filters, with parameters m, n, and k (see section III-B). Assuming the hash functions are perfectly random[4], the probability that a specific bit is 1, after all of the n elements have been inserted into the filter is

$$p = 1 - \left(1 - \frac{1}{m}\right)^{kn}. \tag{9}$$

[3]To form an aggregate, patterns are bit-wise ORed instead of introducing don't cares in bit-positions where they disagree.

[4]A hash function is perfectly random if the hashed value is uniformly distributed over the range, in this case $\{0, m-1\}$

The probability of false match depends on the amount of aggregation, which in turn increases as we move up along the indexing hierarchy. We can estimate the probability (ϕ_l) that an aggregate D (at level l) is a false match for a query Q as:

$$\phi_l = (1 - p^{\tau_l})^{y_l} \tag{10}$$

Here, y_l is the average number of patterns contained in D, and $\tau_l = |Q^1 \wedge D^X|$. Hence, p^{τ_l} is the probability that all of the τ_l 1-bits of Q are present in a pattern contained in D, and ϕ_l stands for the probability that none of the constituent patterns in D, has all of the τ_l 1-bits of Q.

We can estimate y_l as $y_l = \frac{1}{A^l}$ and τ_l as $\tau_l = \tau\left(\frac{m-O}{m}\right)\left(\frac{l}{h}\right)$. Here A and O are as defined in section III-D. $\tau = \chi p m$ is the expected number of ones in Q, and χ is the percentage of elements (*i.e.*, n) from a pattern hashed into Q. $\left(\frac{m-O}{m}\right)\left(\frac{l}{h}\right)$ is the proportion of don't care bits in an aggregate at level l, considering that the number of don't care bits increases linearly as we move up along the indexing hierarchy.

For bitwise-OR based aggregation scheme, we can estimate false match probability as,

$$\psi_l = (1 - p^{\tau})^{y_l} \tag{11}$$

Comparing (10) and (11), we can infer that $\phi_l < \psi_l$ as $\tau_l < \tau$.

C. An Estimate of ξ

Let ν be the probability that a query will fail to match any aggregate at level l, though it matched an aggregate at level $(l+1)$. Then a query will fail to match an aggregate at level l with probability $(1-\nu)^{h-l}\nu$ and in that case $(h-l)$ hops will be wasted. Hence the expected number of probes in a complete descending phase (*i.e.*, from level h to level 0) is:

$$\xi h = h + \sum_{l=0}^{h-1}(h-l)(1-\nu)^{h-l}\nu$$

Applying equality $\sum_{t=0}^{n} t x^t = \frac{x-(n+1)x^{n+1}+nx^{n+2}}{(1-x)^2}$ yields:

$$\xi = 1 + \frac{1}{\nu h}\left[1 - \nu - (h+1)(1-\nu)^{h+1} + h(1-\nu)^{h+2}\right] \tag{12}$$

An estimate of ν is, $\nu = \left(1 - p^{\Delta\tau}\right)^y$. Here, $\Delta\tau = \tau_{l+1} - \tau_l = \frac{p\chi(m-O)}{h}$ and $y = \frac{1}{A}$. y is the average number of aggregates from level l that are fused to form an aggregated at level $(l+1)$.

D. Advertisement Overhead

In this section we compare the advertisement overhead in DPMS against that in DHT-based systems. For DPMS, number of advertisement messages in one refresh interval is $C_{DPMS} = \sum_{l=0}^{h-1} Rn_i = RN\frac{\alpha^h-1}{\alpha^{h+1}-1}$

Let $\Re$ be the total number of advertised patterns in the system. In DPMS $\Re$ can be computed as $\Re = Pn_0 = PN\frac{\alpha-1}{\alpha^{h+1}-1}$. Here, P is the average number of patterns advertised by a leaf peer.[5] Now if this same number of patterns (*i.e.*, $\Re$) are advertised in a DHT-based system, then we can estimate the number of advertisement messages as $C_{DHT} = \Re \lg N = PN\frac{\alpha-1}{\alpha^{h+1}-1}\lg N$. Hence, the ratio of message count in these two systems is:

$$MC = \frac{C_{DPMS}}{C_{DHT}} = \frac{R(\alpha^h - 1)}{P(\alpha-1)\lg N} \tag{13}$$

Assuming W to be the width of a pattern, we can estimate total advertisement volume for DHT-based systems as, $V_{DHT} = \Re W \lg N$. On the other hand, total advertisement volume in DPMS is, $V_{DPMS} = \sum_{l=0}^{h-1} PW(BA)^l n_l R = \Re W R\frac{(RA)^h-1}{RA-1}$. Hence, the ratio of message volume in these two systems is:

$$MV = \frac{V_{DPMS}}{V_{DHT}} = \frac{R\left((RA)^h - 1\right)}{(RA-1)\lg N} \tag{14}$$

In Fig. 5(a) and Fig. 5(b) we produced plots of MC and MV, respectively. For these plots we have used $A = 0.6$, $h = 5$, $P = 10$ and $R = 2, 3$. These are the parameter settings used in the experiments in section V as well. To count for the varying number of peers in the system we have varied B from 4 to 9, which corresponds to a population of 24 to 910 thousand peers for $R = 2$ case and 40 to $1,061$ thousand peers for $R = 3$ case. From Fig. 5 we can infer that advertisement message count in DPMS is much lower than that in DHT-based systems. Advertisement message volume, on the other hand, in DMPS is comparable to that of DHT-based systems for $R = 2$ case, though it is about 4 times higher for $R = 3$ case.

V. Experimental Evaluation

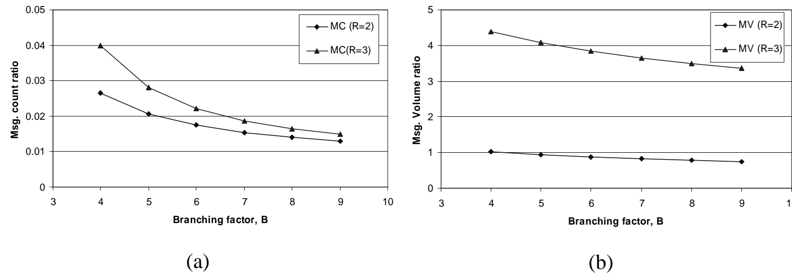
We have conducted experiments for two cases: a) **random case**: patterns are randomly generated bit strings, and b) **biased case**: patterns are Bloom-filters generated using 3-grams from <song title, artist> tuples. These tuples are chosen randomly from a database of 46,500 real-world song information extracted from *http://www.leoslyrics.com/*. A query in random case is created by randomly taking $\frac{1}{3}$ of the 1-bits from a randomly chosen advertised pattern. While in biased case, a query is created as a bloom-filter constructed from $\frac{1}{3}$ of the advertised 3-grams from a randomly chosen advertisement.

We have evaluated routing performance and storage overhead for various parameter settings (section V-A), for different network sizes (section V-B), and for varying levels of replication (with peer failure) (section V-C). We have calculated each data point by averaging the statistical values obtained from independent simulation runs and 3000 queries per simulation run. Each simulation run corresponds to a randomly generated instance of the system.

A. Parameter Tuning

Table II summaries the system parameters and their value(s). Experiments in this section are dedicated for analyzing the impact of different system parameters on query routing performance and storage overhead, and not on fault-tolerance characteristics of the system. Hence, we have chosen R=1. The performance metrics analyzed in this section are:

[5]Unlike DHT techniques, in DPMS a peer can transmit all of its indices in a single message to a parent. Hence, the number of advertisement messages generated in DPMS does not depend on the number of pattern being advertised.



(a) (b)

Fig. 5. Advertisement overhead

TABLE II
SYSTEM PARAMETERS AND THEIR VALUES USED FOR THE PARAMETER TUNING EXPERIMENTS

Param.	Value(s)	Description
B	4	Branching factor
R	1	Replication factor
H	4	Maximum level
N	4092	no. of peers in the system
P	10	number of patterns advertised by a leaf peer.
A	0.6	target aggregation ratio
O	10, 20…60	Minimum number of non-X bits in an aggregate
W	80, 100…200	Pattern or aggregate width, same as m for Bloom-filter

- *First-hit probes* is the number of peers being probed before the first match is found.
- *Avg. probes/hit* is the average number of probes required for each hit. In the cases where multiple matches were present, we traced up to 20 matches.
- *Indexing overhead (IO)* is an indicator for the extra storage space requirement introduced by the indexing hierarchy. IO is measured as:

$$IO = \frac{\text{no. of aggregates (at the indexing peers)}}{\text{no. of patterns (at the leaf peers)}} \quad (15)$$

- *Effectiveness of aggregation (EA)* quantifies the amount of reduction in (index) storage requirement achieved with the aggregation mechanism, and is measured as:

$$EA = 1 - \frac{\text{no. of aggregates with aggregation}}{\text{no. of aggregates without aggregation}} \quad (16)$$

Analyzing the curves in Fig. 6, we can infer the following:

- Query routing accuracy increases with increase in O (Fig. 6(a) and 6(b)) but at the expense of increased storage overhead (see Fig. 6(c) and 6(d)). Information content of the aggregates increases with increase in O, hence better routing efficiency. On the other hand, increase in O implies less space for aggregation, and higher number of aggregates in the system, hence increased storage overhead.
- For $O = W$ and $O = 60|_{W=80}$ cases no aggregation takes place and $\xi = 1$, which gives the upper bound on routing performance and lower bound on storage overhead.
- Not all bits of a pattern are required for query routing with high accuracy. Query routing accuracy is almost the same for $O = 50$ and $O = W$ cases, whereas, indexing overhead for $O = 50$ case is only 65% of $O = W$ case.
- For $O = 10$ and $O = 20$ curves query routing accuracy increases with increase in W. This is because we are using both positional value and content of each bit while matching a query to an aggregate. When W increases, number of possible positions of non-X bits increases, which reduces the probability of false matches. Hence, the decrease in the number of hops.
- For a fixed value of O, IO decreases with increase in W. Given two random patterns, the probability that they will match on a given number of bits increases with W. This results into higher probability of aggregation and lower indexing overhead.

The observations presented for the random case (Fig. 6) are equally applicable for the biased case (Fig. 7). In addition, we can infer the following by comparing these graphs.

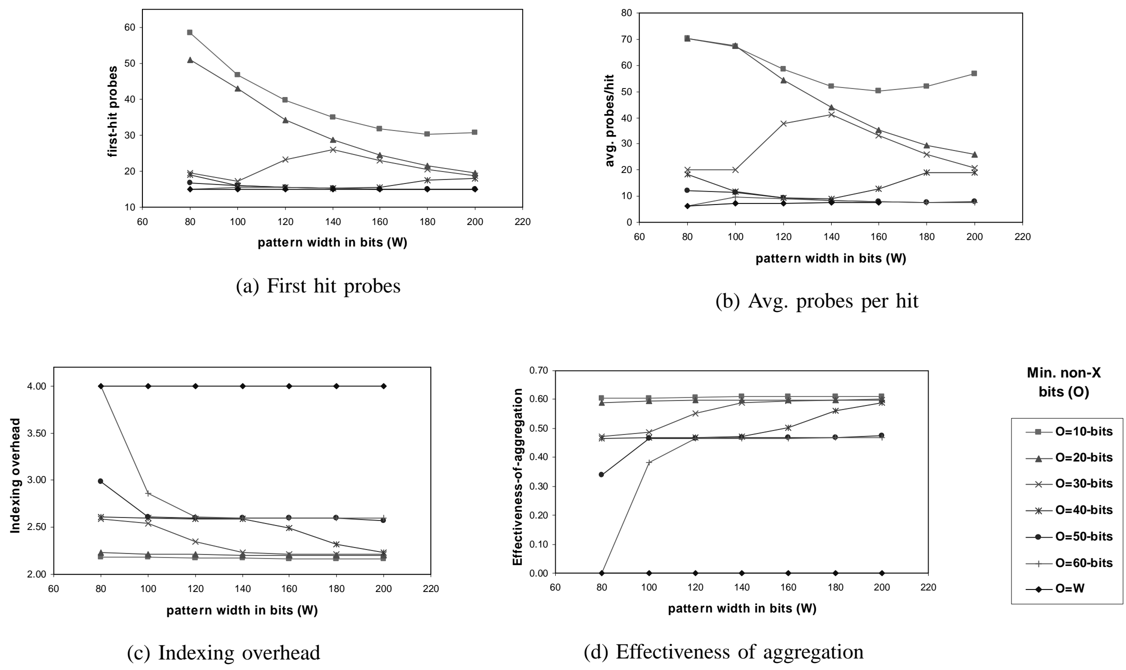
- Query routing performance is in general better for the biased case. Specially for $O = 10$ and $O = 20$ curves it is about 3 times better, while EA is almost identical.
- IO and EA are better for the biased case as well. For $O \leq 40$ curves, EA reaches its upper limit [6] for the biased case (Fig. 6(d) and Fig. 7(d)). Which implies, higher level of aggregation is possible by reducing A, without sacrificing routing performance.

B. Scaling Behavior

In this section we analyze the scaling behavior of DPMS with growth in network size. Based on the observations in section V-A, we have chosen $W = 180$ and $O = 50$ for the experiments presented in this section. N has been varied from around $8,000$ to $21,000$ while keeping the number of peers per group at the highest level of the indexing hierarchy in the range of $0.6 \log N$ and $1.5 \log N$. We have used $R = 1$ and $B = 4$ and the value of H was set to 5 to accommodate all the peers in the indexing hierarchy, without violating the above mentioned constraints.

In Fig. 8, the *1st hit Probes (estimated)* curve is a plot of $probes = f \log_B N + \xi_{est} * h_{est}$.

[6] Upper limit for $EA = 1 - \frac{\sum_{l=1}^{h} PA^{l-1}B^l n_l}{\sum_{l=1}^{h} PB^l n_l} = 1 - \frac{A(A^h-1)}{h(A-1)}$ (note: leaf peer do not aggregate and $R = 1$)



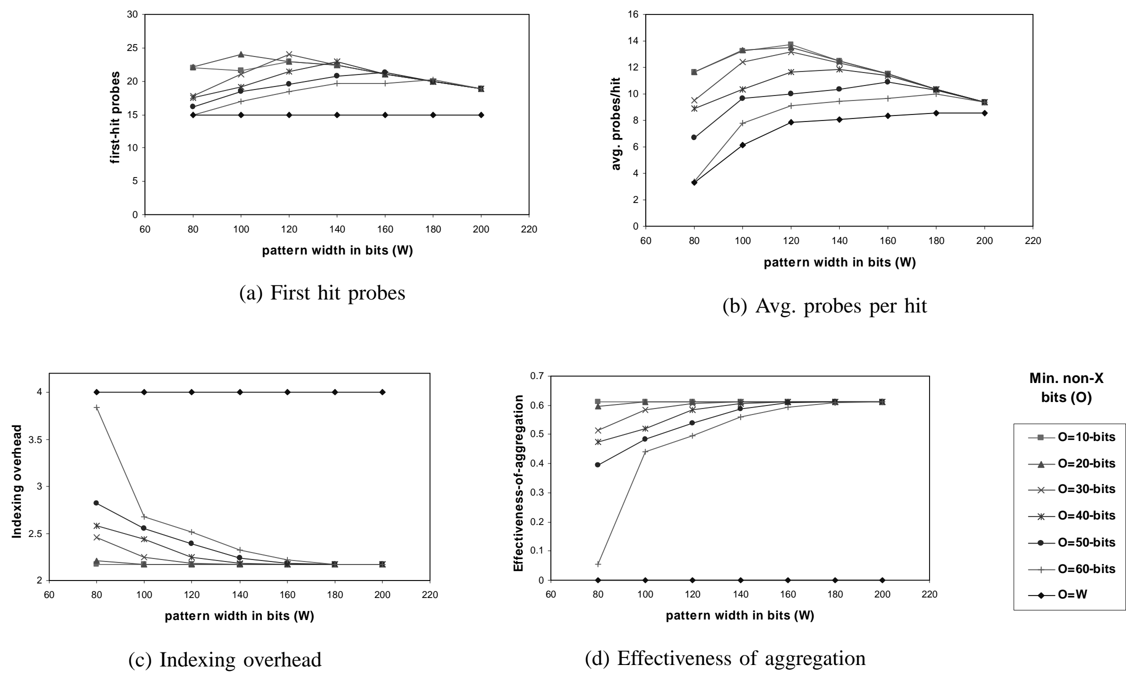
(a) First hit probes

(b) Avg. probes per hit

(c) Indexing overhead

(d) Effectiveness of aggregation

Fig. 6. Impact of O and W on routing performance and storage overhead (random case)



(a) First hit probes

(b) Avg. probes per hit

(c) Indexing overhead

(d) Effectiveness of aggregation

Fig. 7. Impact of O and W on routing performance and storage overhead (biased case)

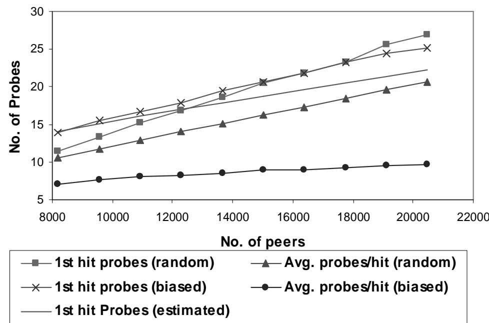


Fig. 8. Scaling behavior

ξ_{est} has been derived from (12) using the parameter values used in this experiment and putting $\chi = 0.33$ [7]. h_{est} has been derived from (7) for $R = 1$ as, $h_{est} = \log_B \left(\frac{1}{B} + \frac{N(B-1)}{Bf \log_B N} \right)$.

It is evident from Fig. 8 that the routing efficiency a) does not decrease significantly with network growth, b) is much better for biased case than that of the random case, and c) is close enough to the estimated curve.

For this experiment none of the parameters affecting storage overhead where varied, hence IO and EA were constant. IO and EA for random case were 4.04 and 0.24, respectively; and for biased case were 3.07 and 0.48, respectively.

C. Fault-tolerance

For the experiments in this section we have varied R from 1 to 6 while keeping all other parameters at a constant value. We have used $H = 4$, $B = 4$, $O = 50$ and $W = 180$. N was varied from about $4,000$ ($R = 1$) to $40,500$ ($R = 6$). For each value of R we have deactivated (*i.e.*, removed) up to 50% of the peers in 5% step, and have executed 3000 queries on the rest of the peers. The impact of bias among the patterns is orthogonal to fault-tolerance characteristics of the system, hence we have presented only the random case in this section.

Failure of indexing peers may result into unreachable leaf peers (from level H). To measure the impact of this phenomena we have defined *hit rate* (Fig. 9(a)) as the average percentage of matches that are discovered by a query. The impact of replication on the overall storage overhead in the system is presented in Fig. 10.

By analyzing the curves in Fig. 9 and Fig. 10 we can infer the following:

- Without replication ($R = 1$) hit-rate and routing efficiency decrease drastically as more and more peers fails. In this case failure of each indexing peer results into many unreachable leaf peers and increases false match rate.
- The downside of replication is the exponential growth in IO (see Fig. 10(a)). However EA increases with increases in R (see Fig. 10(b)). It can be shown that the value of EA tends to $1 - A^{h-1}$ as R tends to infinity. This justifies the curve in Fig 10(b).
- For resilient query routing, replication is necessary. But a small value of R (*e.g.*, 2 or 3) would suffice this need for most systems, assuming up to 30% failure rate.

[7] $\frac{1}{3}$ of the elements present in a randomly selected <song title,artist> tuple was used for generating a query

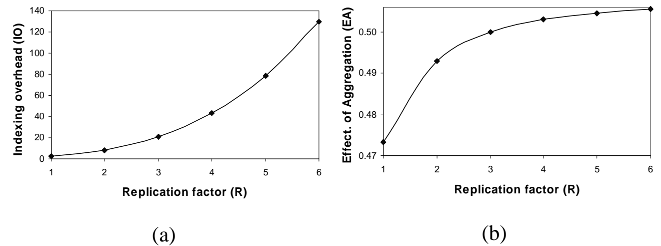


Fig. 10. Impact of replication on storage overhead

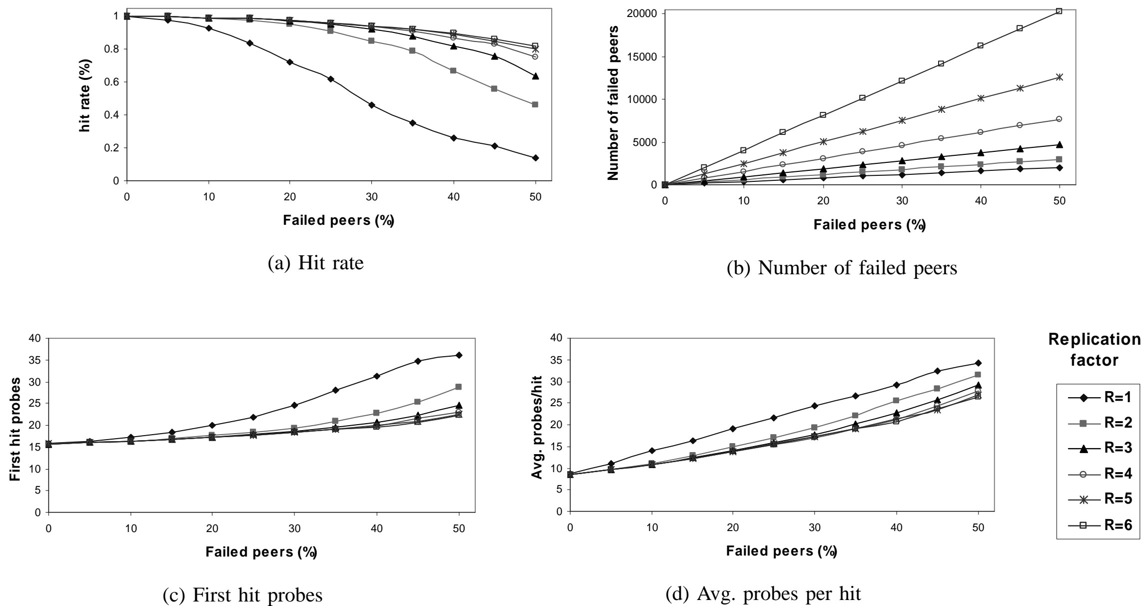
VI. Conclusion and Future Works

In this paper we have defined the Distributed Pattern Matching (DPM) problem and have presented a scalable solution, DPMS, to this problem. DPMS supports flexible queries involving partial and multiple keywords. Query routing efficiency of DPMS is comparable to that of the structured P2P systems. For moderately stable networks, DPMS provides guarantee on search completeness and on the discovery of rare items. Peers in DPMS maintain constant number of links, in contrast to $O(\log N)$ links per peer required by most DHT-based systems. DPMS can easily be tailored to achieve context-sensitive (*e.g.*, network proximity, user interest *etc.*) query routing. Moreover, DPMS can exploit the heterogeneity in peer capabilities, and does not place any hard restriction on the placement of documents or indices.

The main drawback of the proposed system is the storage overhead introduced by hierarchical indexing and replication. Experimental results presented in this paper demonstrate the worst possible values for the storage overhead (*i.e.*, for random case). For most applications, there exists some bias among the advertised patterns, which can enable higher levels of aggregation and hence lower levels of storage overhead, as demonstrated by the results for the biased case. Another problem in DPMS stems from leave/join of leaf peers. Leave/join of indexing peers has local effect only. But, leave/join of a leaf peer results into cascaded updates along its replication tree. This problem can be mitigated by using periodic and differential updates of index information between adjacent indexing peers. This latency in update will not hamper the normal operation of the system other than degrading query routing performance to some extent.

Modification of data at leaf peers (*e.g.*, change in filename, insertion of new data/file or deletion of existing data/file) will invalidate the associated index information. This problem persists in any structured or semi-structured system, though the effect is higher in a hierarchical indexing system like DPMS. It is possible to reduce the effect of data dynamism using periodic index updates. Another possible measure is to allow approximate matching in query pattern and indexed pattern/aggregate instead of strict subset matching.

We are extending this research to demonstrate the applicability of DPMS in different application domains, including service discovery and P2P databases. We also intend to investigate the self-tuning and self-optimization aspects of DPMS, as our future research in this direction. Other possible extensions to this research include, a) incorporating context-sensitive routing in DPMS and measuring its effectiveness,



(a) Hit rate

(b) Number of failed peers

(c) First hit probes

(d) Avg. probes per hit

Fig. 9. Impact of replication and peer failure on routing performance and hit rate of a query

and b) enhancing DPMS by adopting Bloomier filters [8] for supporting range queries.

REFERENCES

[1] Fasttrack peer-to-peer technology, http://www.fasttrack.nu/.

[2] Gnutella website, http://www.gnutella.com.

[3] R. Ahmed and R. Boutaba. Distributed pattern matching for p2p systems. In *Proc. of NOMS'06 (to appear)*, Online [http://www.cs.uwaterloo.ca/ r5ahmed/DPMS-noms06.pdf].

[4] A. Amir, E. Porat, and M. Lewenstein. Approximate subset matching with don't cares. In *Proc. of SODA*, pages 305–306, 2001.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. of ACM*, 13(7):422–426, 1970.

[6] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

[7] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like P2P systems scalable. In *Proc. of ACM SIGCOMM*, pages 407–418, 2003.

[8] B. Chazelley, J. Kilianz, R. Rubinfeldz, and A. Tal. The bloomier filter: An efficient data structure for static support lookup tables. In *Proc of SODA*. 2004.

[9] E. Cohen, A. Fiat, and H. Kaplan. Associative search in peer to peer networks: Harnessing latent semantics. In *Proc. of IEEE INFOCOM*, 2003.

[10] R. Cole and R. Harihan. Tree pattern matching and subset matching in randomized $o(n \log^3 m)$ time. In *Proc. of ACM STOC*, pages 66–75, 1997.

[11] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An Architecture for a Secure Service Discovery Service. In *Proc. of MOBICOM*, pages 24–35, 1999.

[12] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based Peer-to-Peer networks. In *Proc. of IPTPS*, pages 242–259, 2002.

[13] M. Li, W. Lee, and A. Sivasubramaniam. Neighborhood signatures for searching p2p networks. In *Proc. of IDEAS*, pages 149–159, 2003.

[14] L. Liu, K. D. Ryu, and K. Lee. Supporting efficient keyword-based file search in peer-to-peer file sharing systems. In *Proc. of GLOBECOM*, 2004.

[15] C. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. of ICS*, 2002.

[16] P. Maymounkov and D. Mazi. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. of IPTPS*, pages 53–65, 2002.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, pages 161–172, 2001.

[18] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. of IEEE INFOCOM*, 2002.

[19] C. Schmidt and M. Parashar. Enabling flexible queries with guarantees in p2p systems. *IEEE Internet Computing*, 8(3):19–26, 2004.

[20] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. on Networking*, 11(1):17–32, 2003.

[21] D. Tsoumakos and N. Roussopoulos. Adaptive probabilistic search for peer-to-peer networks. In *IEEE Intl. Conf. on P2P Computing*, 2003.

[22] S. Voulgaris, M. Jelasity, and M. van Steen. A robust and scalable peer-to-peer gossiping protocol. In *Proc. of the Intl Workshop on Agents and Peer-to-Peer Computing*, 2003.

[23] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *Proc. of ICDCS*, 2002.

[24] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE JSAC*, 22(1):41–53, 2004.