Manuscript received Nov. 11, 2007; revised Apr. 12, 2008

A Protocol For Self-Organizing Peer-to-Peer Network Supporting Content-Based Search

IGOR MEKTEROVIĆ, MIRTA BARANOVIĆ, KREŠIMIR KRIŽANOVIĆ
Department of Applied Computing
Faculty of Electrical Engineering and Computing, University Of Zagreb
Unska 3, 10000 Zagreb
CROATIA
igor.mekterovic@fer.hr http://www.zpr.fer.hr/zpr/people/igor/

Abstract: - For a peer-to-peer(P2P) content sharing network holding large amount of data, an efficient semantic based search mechanism is a key requisite. Semantic based search should generate as little traffic (messages) possible while achieving precision and recall rates comparable to those of correspondent centralized system. In this paper protocols for self-organizing P2P networks that arranges links between peers according to peer's content are developed and tested. Peers organize themselves into "semantic communities" without losing links to other semantic communities. Proposed network requires no prior knowledge of the semantics of documents that are to be shared in the system. Through simulations, it is shown that proposed network is resilient to membership changes and achieves high recall rates.

Key-Words: - Peer-to-peer, Content-based search, Information retrieval, Algorithm, Semantic

1 Introduction

With the advent of Napster we have witnessed extraordinary expansion of interest in peer-to-peer content sharing systems both in general population and in scientific community. With time, better and more efficient protocols (networks) for content sharing have been developed (e.g. Gnutella, eDonkey, BitTorrent). However, widely adopted peer-to-peer protocols for content sharing only allow for metadata searches (file name, size, type, etc.). Peer-to-peer networks that enable content-based searches are still a subject of active research. The ones that have been developed so far are usually divided into structured and unstructured. In unstructured networks peers are unaware of content in neighboring peers which coerces them into less-effective query routing (e.g. Gnutella flooding) resulting with poor network scalability. Structured P2P networks overcome scalability issues but incur complex protocols that are not suitable for highly transient peers typical for P2P systems [13]. Also, structured P2P networks usually have to maintain high-dimensional DHTs (Distributed Hash Tables) that reflect semantic space of documents stored in the system. Such DHTs may be inappropriate for the newly arrived documents whose semantics significantly differ from those of documents that were taken into account during construction of DHT. Along those lines, if we wanted to construct an initially empty network and offer it to the general public to share arbitrary documents we'd have no documents to sample and construct the semantic space.

That was the motivation for construction of P2P network that allows for semantic-based queries without prior knowledge of documents that will be stored throughout the network. Self-organizing P2P network that arranges links between peers according to their content is proposed. In such a network peers organize themselves into "semantic communities". Every peer represents its content with a set of vectors and content likeness is determined as vector likeness. It is assumed that peers (i.e. users) sharing documents of certain topic will most likely search for similar documents (e.g. someone who is sharing papers in the field of computer science is more likely to search for similar papers than e.g. biology papers). Of course, it is entirely possible for user to search for something semantically completely different – therefore it is important not to lose links to other semantic communities.

2 Problem Formulation

Textual documents can be represented and stored as data objects in P2P system. More precisely, a document is represented as an n-dimensional vector, namely Semantic Vector or Feature Vector. Each element in the vector represents the importance of a term in the document, usually computed using TF*IDF (term frequency * inverse document frequency) scheme [1]. A term is considered more important within the document if it is used often in that document (TF) and used seldom in other documents in the collection (IDF). Such term is important because it differentiates one document from

the others. During the search process, documents are retrieved according to the similarity between the query vector (which can also be a full-blown document) and document vector. Prevailing measure of similarity is the cosine of the angle between the vectors. If the vectors are normalized, cosine of the angle can be computed as the inner of product of two vectors:

$$\cos(Q,D) = \frac{Q \circ D}{|Q| \cdot |D|} = \sum_{i=1}^{n} q_i \cdot d_i \qquad (1)$$

This model, in which documents are represented as vectors, is referred to as Vector Space Model (VSM). VSM suffers from synonymy, polysemy and noise in the documents. LSI (Latent Semantic Indexing) [2] technique has been proposed to overcome these issues. LSI uses SVD (singular value decomposition) [1] to transform a high-dimensional VSM vector to a lower-dimensional semantic vector by projecting it into a smaller, semantic, subspace. In summary, both VSM and LSI represent documents as vectors and use cosine of the angle between the vectors to represent their similarity.

Searching for a document in a P2P environment could be done in Gnutella fashion by flooding the neighborhood with query vector, however that approach has been proven to suffer from scalability issues. Also, since documents are randomly populated (with respect to semantics) it is difficult to achieve good retrieval properties (precision and recall). Efforts to improve the search efficiency have led to constructing structured overlay networks (e.g. CAN [3], CHORD[4], Tapestry[5], Pastry[6]). These systems support hash-table interface of *put(key, value)* and *get(key)* and are extremely scalable as they resolve lookups in log(n) routing hops (for a network of n nodes). On the other hand, they support only exact-match queries. Since then, more sophisticated structured systems have been developed (e.g.[7], [8], [9]) that are both scalable and allow for semantic queries. However, being structured, they all have to form a semantic space, probably (not all papers explain it) by sampling documents that are expected to be shared in the P2P network. Although they work well under such conditions, we believe that this presents a problem in the case when there is no prior knowledge of semantics of documents that will be shared throughout the network. In this article the possibility of creating a network that will not require prior knowledge of content to be shared is explored.

3 Related work

In general, there are two strategies for performing search in P2P network: (a) blind search where nodes "blindly" propagate messages to, hopefully, sufficient number of other nodes and (b) informed search where nodes use local information about neighboring nodes to route messages towards (estimated) relevant nodes. Gnutella is the best-known blind search method. Other blind search methods include Modified-BFS [14] where peers randomly choose only a ratio of their neighbors to forwards query to, Iterative Deepening [16] and Random Walks [15]. The latter two work well only when it is not required to find all relevant documents in the network, but, typically, only one. That makes them unsuitable for the problem studied here. Informed search methods include APS [17], LI [16], RI [18] and PlanetP[19]. APS builds upon an idea of random walks but instead of walks being random it uses probabilistic forwarding based upon statistics that is accumulated in time. In LI every node indexes the content of neighboring nodes (within some radius r) and answers the queries on their behalf. This approach is not well suited for dynamic environments. RI assumes that all documents fall within a number of thematic categories and each node stores an approximate number of documents that can be reached through each of the outgoing links and routes queries in accordance. This approach works well only for some applications. In PlanetP nodes use Bloom filters to create compact representations of their inverted indexes and then diffuse them throughout the community using gossiping algorithm. Since it doesn't scale well, PlanetP is suitable only for small and mid size networks. A thorough overview and comparison of unstructured P2P search methods can be found in [20].

Structured networks (e.g. [7], [8], [9]) use a different approach and try to combine the advantages of structured systems in order to achieve better search efficiency and scalability properties. However, they tend to be significantly more complex which complicates their deployment in dynamic environments. Also, they require some prior knowledge of the semantic of the data to be shared throughout the network.

4 Self-organizing network

Both pure (Fig.1) and hybrid (Fig.6) P2P networks are presented.

4.1 Pure network

Besides sharing content, every node in the pure network routes messages through the overlay network and exchanges overlay network maintenance messages (Fig.1). That is, there is no hierarchy of nodes or nodes performing special functions.

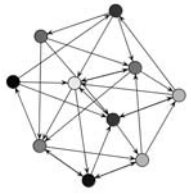

Fig.1 Pure network of 10 nodes

Every node represents its content (documents) with semantic vectors (VSM). Vectors can be computed using global statistics that, as demonstrated in [10], doesn't have to be precise. To reduce the number of vectors similar documents are clustered together and represented with cluster centroid. Number of clusters (or documents per cluster) is arbitrary – it can be tuned over time or even left up to the user to decide (e.g. a user could mark the spot on the dendrogram) although it shouldn't be too high (this will be a topic for future research). Such set of vectors representing cluster centroids is called node description. In order for a node to join the network it has to connect to existing node(s). Since there is no central authority a node has to find those nodes on its own. In our research new nodes were connected to random existing nodes in the network, and in the real-word applications some already existing techniques (like GWebCaches in Gnutella) could be employed. Each node maintains two sets of links: *family links* and *others links*. *Family links* are used as a connection to other nodes with similar description and *other links* are used as a connection to communities of dissimilar nodes. Figure 2 shows separately *other links*, *family links* and all links for a network of 50 nodes and 3 semantically very distant communities.

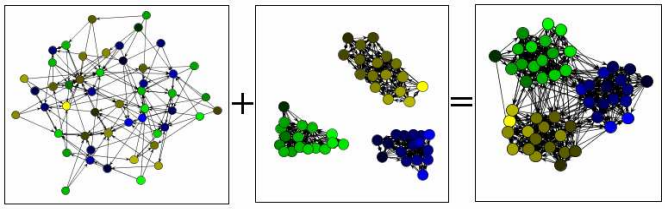

Fig.2 *other links* + *family links* = all links (50 nodes)

Initially, each node only has *other links*. Node will start to populate its *family links* collection when it receives answers to its queries.

4.1.1 Query routing

When a node wants to search for a content in the network it creates a *QueryMessage* and routes it through the network according to algorithm in Table 1. *QueryMessage* consists of:

- unique message id
- source peer id (node who originated the message)
- previous peer id (node that forwarded this message)
- query vector (describing sought content)
- similarity threshold (for determining the results)
- pair of bloom filters

Two bloom filters are used to reduce the number of messages that are transmitted through the community of nodes that match the query (based on the similarity to the query vector). Fig.3 shows a query routing scenario without the use of bloom filters in a small community of interconnected nodes (links are not drawn for clarity). Since every node maintains a list of processed queries, all messages carrying already processed queries are dropped. Dropped messages are drawn with a dotted line. A scenario is shown in which every node can forward maximum two messages and node *f* never gets the message because nodes *d* and *e* are unaware that nodes *c*, *e* and *d* have already gotten the message. To improve message routing two bloom filters [12] are added to the message. Whenever a node forwards the query it embeds into the message node ids (hashes) of all nodes that it will be sending the message to. Accordingly, when a message is forwarded bloom filter is checked to determine whether a node already received the message. Fig. 4 shows the worst case query routing scenario with the use of single bloom filter. Associated table details information about visited nodes that is carried in the correspondent message (e.g. *bd:bcde* means that message from node *b* to node *d* has nodes *b,c,d* and *e* defined as visited nodes). Of course, the probability of false positives increases as the number of inserted elements increase. We've split bloom filter into two bloom filters: they are populated one after another (with possibility of false positives below 8%) and when the second bloom filter is full the first one is cleared assuming that the query has "moved away" from the area recorded in the first bloom filter (this doesn't have to be true). When communities with 100-400 nodes, 20 links per node and maximum forward count 10 were flooded it was found that two 100-bit bloom filters have reduced the number of messages by more than 50%.

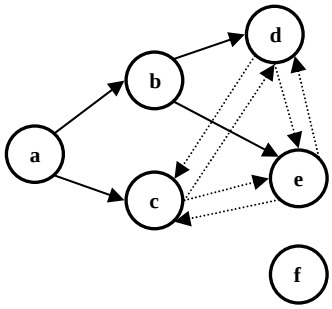

Fig.3. Routing without bloom filter, MAX_FW_CNT=2

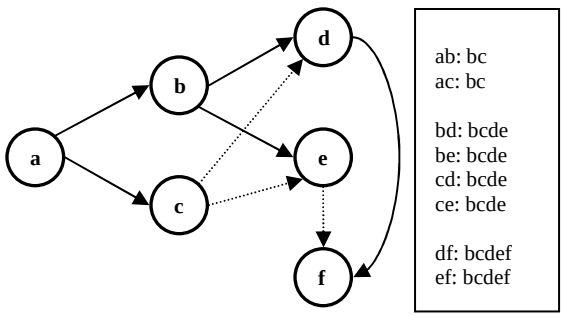

Fig.4. Routing with bloom filter, MAX_FW_CNT=2

As shown in Table 1, node forms a descending list of other nodes based on their similarity to the query vector. Similarity threshold is embedded in the message and, if the query returns too few results, could be adjusted by user (application) to broaden the search..

```
routeQuery(qMsg)
  FwList = getFwList(qMsg)
  IF qMsg.notDoneHopping()
    qMsg.setNodesVisited(FwList)
    qMsg.setPreviousNode(this)
    forward query to every node in the FwList
  END IF

getFwList(qMsg)
  FW = getRankedNodesDesc(qMsg, qMsg.sim)
  IF FW.size > MAX_FW_CNT
    FwList = pick random MAX_FW_CNT
                peers from FW
  ELSE IF FW.size < MIN_FW_CNT
    qMsg.decTTL()
    FW = getRankedNodesDesc(qMsg.query, 0)
    FwList = get first MIN_FW_CNT peers from FW
  ELSE
    FwList = get all peers from FW
  END IF
  RETURN FwList

getRankedNodesDesc(qMsg, simTreshold)
  FO = Family U Others
  FW = Ø
  FOREACH currNode IN FO
    IF currNode≠qMsg.src AND currNode ≠qMsg.prev
       AND qMsg.notVisited(currNode)
      IF curr.emptyDesc
        FW = FW U (curr, 0)
      ELSE IF qMsg.queryVector.getSimilarity(curr)
                ≥ simTreshold
        FW = FW U (curr,
                qMsg.queryVector.getSimilarity(curr))
      END IF
    END IF
  END FOREACH
  FW.sortDescending()
  RETURN FW
```

Table 1. Routing algorithm for pure network

When a node has finished evaluating similarity to the query vector, it compares the query vector both to the *family links* and *other links* collection. That way, if a query has reached targeted semantic community probably only nodes from the *family links* collection will be used to forward the query (depending on the threshold and the community a query could even be flooded through the community). On the other hand, if the query is somewhere outside the targeted community then probably a most similar node will be found in the *other links* collection – hopefully that link will lead to the desired community. If none of the known nodes satisfies the threshold requirement then a minimum forwarding rule is activated: query is forwarded to MIN_FW_CNT (e.g. MIN_FW_CNT=1) nodes disregarding the similarity threshold but message's TTL (time to live) attribute is decreased by one. Thus, a message has only TTL hops to reach the targeted community (probably through a series of *other links*) but once inside the community TTL value doesn't change. On the other hand, if a node computes that more than MAX_FW_CNT links (nodes) meet the threshold requirement then a maximum forwarding rule is activated: query message is forwarded to randomly picked MAX_FW_CNT nodes from the set of nodes that satisfy the threshold requirement. Initially, message was forwarded to MAX_FW_CNT most similar nodes but that strategy has been shown to favor only the most similar nodes ignoring the less similar nodes that also meet the threshold requirement thus reducing the recall

Besides routing (forwarding) query message every node evaluates its collection of documents against query vector. If any of the documents meets the threshold the node sends a *QueryResponse* message to the node that originated the query. *QueryResponse* message carries the following information:

- query message id
- responder peer id (peer who responds to the message)
- responder peer description (cluster centroids)
- response vectors (documents that match the query)

Every time a node receives a *QueryResponse* message it updates its *family links* collection with the responder's node description: nodes are sorted descending based on the similarity with its own description. *Family links* collection size is limited and if it exceeds the maximum then a randomly picked node from the bottom N percent (e.g. 10%) of the list is removed. Node maintains its *other links* collection using *MeetTheOthers* message that it emitts from time to time. *MeetTheOthers* message is randomly forwarded through the network. *MeetTheOthers* message has:

- source peer id
- TTL (decreased with every hop)
- visitedNodes[TTL] array

When a *MeetTheOthers* message is instantiated, a TTL value is randomly chosen from a predefined interval. Since this is a small number of hops an array of visited nodes is carried by the message to avoid reaching a same node twice. Every node that receives *MeetTheOthers* message responds with *NodeDescription m*essage (carrying only its own description) to source node and then, if TTL is still greater than zero, forwards received message to only one randomly chosen node (that hasn't been visited) from the *other links* collection.

Upon receiving *NodeDescription* message a node updates its *other links* collection and if it exceeds the maximum size removes randomly chosen old entry. Node description is not taken into account (unlike with *family links* collection) when a node is removed from *other links* collection – node descriptions of other nodes are only used in the process of query routing.

In the event of node failure, nodes simply discard the links with the nodes that don't respond to their messages and other nodes replace the disconnected ones.

4.1.2. Membership changes

Since P2P networks are inherently transient it is important for them to handle peer join and peer leave (or fail) operations gracefully. When a peer joins a pure network using some bootstrap node (in our simulations a randomly chosen node is used) it simply emits a *MeetTheOthers* message with TTL equal to the *other links* collection size. That way, new node gets "wired" into the network and gradually forms family links i.e. positions itself within the community. When a newly arrived node receives an answer to its query it considers a replier to be family and sends its own description to the replier (if it doesn't already know it – replier includes a description version in his reply). That way existing nodes form their links to the newly arrived node. Pure network shows resilience to peer failures. A node detects a "dead" link when it fails to send a message to another node. In that case the node simply deletes dead link from the collection and resends the message to someone else. If, by doing so, the number of links decreases below a certain threshold node emits a *MeetTheOthers* message. Fig. 5 shows network properties when 60% of the nodes leave the network: between iterations 10 and 20, 3000 nodes leave the network. In one iteration all nodes in the network query for a content similar to their. Recall (avg_nodes_recall) is defined as a ratio of retrieved relevant documents and relevant documents in the entire network. Average relative message count (avg_rel_msg_count) is defined as query message count divided by network size. Average percentage reached (avg_perc_reached) is the average percentage of nodes reached during a query. Maintenance message count (avg_mntnc_msg_count) is absolute average number of maintenance messages (it is shown on the same graph to conserve space, e.g. avg_mntnc_msg_count for 1st iteration is 21 not 21%). Fig. 5 shows no significant improvement in recall when peers leave the network (and send messages of notification to their neighbors) over the case when peers simply fail. Moreover, nodes that leave incur slightly more traffic. That is why we've decided that in pure network protocol nodes do not inform neighbors of their departure, i.e. all departing nodes are behaving as if failing.

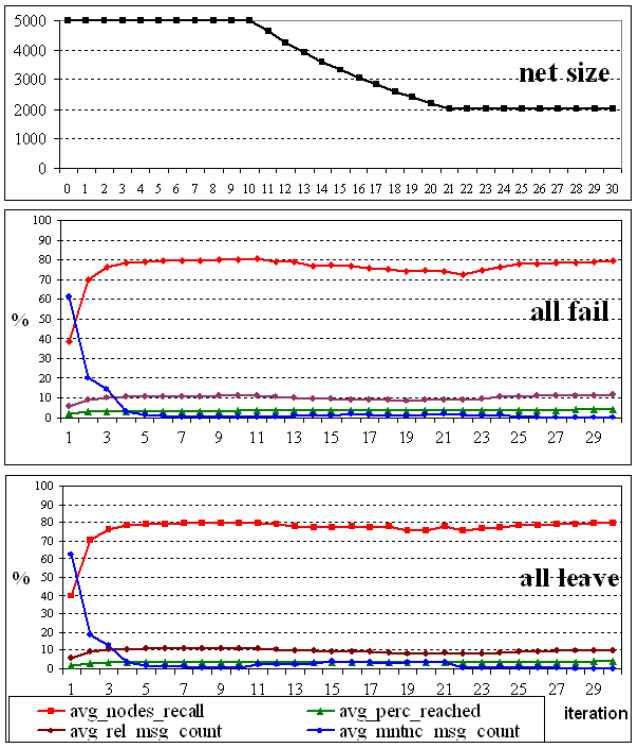

Fig. 5 Effects of 60% nodes failing/leaving the pure network

4.2. Hybrid network

In order to reduce the traffic (and increase scalability) Gnutella designers have switched from the initial pure (version 0.4) to hybrid (version 0.6) architecture. Accordingly, we've developed hybrid self-organizing network that distinguishes two kinds of nodes: leaf and ultra nodes. The idea is to put more capable (in terms of bandwidth, availability and processing power) nodes in charge of routing the query messages and network maintenance messages. Such nodes are called ultra nodes. Every ultra node maintains connections to a certain number of leaf nodes. Leaf nodes send queries to their ultra node and have no role in query routing process. However in our protocol leaf nodes do communicate with other ultra nodes in attempt to cluster themselves in the semantic communities. Fig. 6 shows a hybrid network with 4 ultra nodes, each attending to 4 leaf nodes.

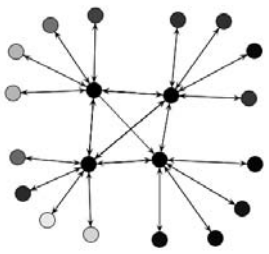

Fig. 6 Hybrid network with 4 ultra and 16 leaf nodes

4.2.1 Bootstrapping

The network is setup by linking few ultra nodes. Other ultra nodes attempting to join the network have to find an existing ultra node(s) to link with. The process is analogous to the one in the pure network. Leaf nodes attempt to join the network by sending a *JoinRequestMessage* (Fig. 7) carrying node description to an ultra node. If the ultra node already maintains maximum leaf connections it replies with a negative *JoinReply* message carrying a list of alternative ultra nodes (e.g. *u2*) the leaf can then try to join. Upon receiving a negative reply the leaf node tries to join another ultra node in the list. If the ultra node hasn't reached maximum number of leaf connections it replies with a positive *JoinReply* message and adds the leaf (node description) to its *leaf links* collection.

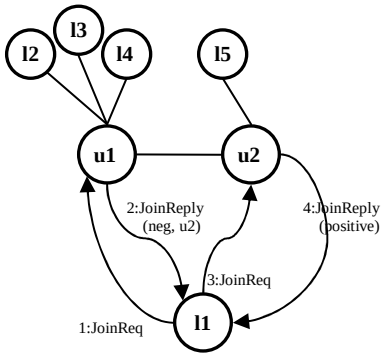

Fig. 7 Leaf node joining the network

4.2.2 Query routing

In hybrid architecture only ultra nodes are responsible for query routing thus shielding the leaf nodes. A leaf node creates a query message and sends it to its ultra node. From that moment on ultra nodes subnet functions analogous to the pure network and routes the query according to the similarity of the known (*family* and *other*) ultra nodes. Besides forwarding the query to other ultra nodes ultra node may forward the query to its leaf. Every ultra node has node descriptions of its leaf nodes and if it finds that a leaf node description is similar enough to the query vector, it forwards the message to that leaf node. Leaf node further examines the query message and if the query matches any of its documents it replies directly to leaf that originated the query. In addition to response vectors *QueryResponse* message carries node description of the responder's ultra node (ultra node description consists of leaf nodes description). This information will be used to cluster similar leafs together.

4.2.3 Leaf migration

Every leaf keeps track of ultra nodes and number of results it received from their leaves. If a leaf is in the right community (after a significant number of queries) the number of results it received from the best other ultra node should be comparable to the number of results it received from its own ultra node. That would mean that neighboring (having the same ultra node) leaf nodes are replying to some of its queries. Otherwise, if node is in a wrong community, it will receive most of its replies from leaves that are not neighboring. In that case, after the leaf node has concluded that there is a better ultra node available (semantically more fitting), leaf node sends a *TransferRequest* message (message 1 on Fig. 8) to the ultra node whose leaves are responsible for most results providing that it cannot be found in the negative transfer attempts cache. Every leaf nodes maintains this cache of ultra node ids that returned negative *TransferReply* messages so that it wouldn't subsequently send the same *TransferRequest* messages to same ultra nodes (this cache is periodically cleared).

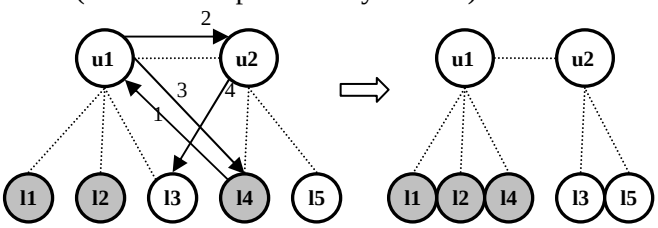

Fig. 8 Leaf transfer with replacement leaf

Ultra node will approve the transfer if:
(a) it hasn't reached its maximum leaf collection size
(b) it has reached its maximum leaf collection size but there is a leaf in the collection that is semantically less befitting to that community than the requesting leaf (we call it the replacement node).
In the latter case, ultra node removes the replacement leaf from its collection and sends a *TransferExchange* message (message 2 on Fig.8) to requester's ultra node (this information is included in the transfer request message) informing ultra node to remove the requester leaf from its collection and add replacement leaf instead (in Fig. 8 – remove *l4* and add *l3*). In the former case (a) *TransferExchange* message doesn't include the replacement node. In both cases, after receiving *TransferExchange* message, ultra node responds to requester leaf node with a positive *TransferReply* message and adds the requester leaf to its leaf collection. Requester leaf updates its ultra node link to the new node. If the requester's old ultra node received *TransferExchange* that included replacement node, it sends a *JoinMe* message to the replacement node (message 4 in Fig. 8) ordering the replacement node to update its ultra node (in Fig. 8 replacement node sets *u2* as the new ultra node). When a leaf node changes its ultra node it resets query response count statistics.

If none of the two conditions are met (a and b) ultra node responds to the requester leaf with a negative *TransferReply* message. In that case, only messages

marked 1 and 2 in the Fig.8 are exchanged. Leaf node that receives negative *TransferReply* message sets the response count of that ultra node to zero and stores its id in the negative transfer attempts cache.

4.2.4 Load balancing

This leaf migration strategy leads to clustering of similar nodes and the more similar nodes there are at some ultra node, the more it becomes attractive for other similar leaf nodes to transfer there. This way, some ultra nodes begin to accumulate more and more leaves (until they are full) and other ultra nodes lose leaves as they migrate to other ultra nodes (stage B in Fig.9).

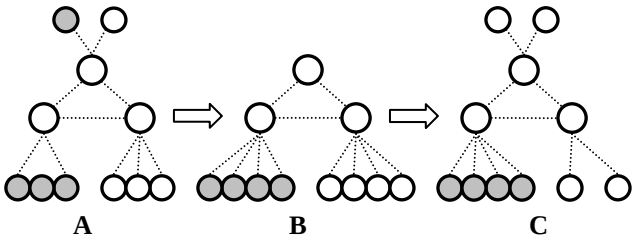

Fig.9 Leaf migration and load balancing

To that purpose a set of messages that will allow ultra nodes to balance the load (leaf connections) is defined. Following variables are defined:

- M – maximum number of leaves
- Lf – number of leaves at balance source node
- Lb – number of leaves at balance destination node
- p – percentage of leaves that balance source node gives away (we use p = 50%)
- T – threshold (if an ultra node has more or equal to M*T leaves then it may be considered as balance source node). We've set threshold at 70%.

When an ultra node has too few leaves (less than Lb) it sends a *BalanceRequest* message to a neighboring ultra node. Lb is determined from the equation (which states that balance destination node should not have more than T*Lf leaves after the balancing process):

$$Lf*p + Lb <= T*M \quad (2)$$

which, if Lf is set to M as the worst case, evaluates to:

$$Lb <= M*(T-p) \quad (3)$$

Therefore, when an ultra node falls down to less than M*(T-p) leaves (i.e. in our simulations less than 20% of M) it starts to send *BalanceRequest* messages. Related to this is the estimation of the similarity factor that a leaf node uses to determine whether to apply for transfer: if a leaf node finds that another ultra node's leaves provide more results than similarity factor times number of results its current ultra node provided, it then applies for transfer. To prevent leaves that have just been balanced to reapply for the old ultra node (or another, more populated one) similarity factor (Sf) is estimated as:

$$Sf *(Lf*p -1) > M \quad (4)$$

stating that a leaf that has just been balanced should be satisfied with a number of results it recieved from its new ultra node (Lf*p-1) even though some other full ultra node is providing more results (M). From equation (4) we get:

$$Sf > 1 / (T*p - 1/M) \quad (5)$$

On the other hand, setting Sf too high would slow up the process rendering network inert. That's why we define [Sfmin, Sfmax] interval and every node starts with a minimum value of Sf (making it more mobile) that is incremented on every transfer until it reached Sfmax (leaf nodes become less mobile as they "grow old"). We use interval [1,7]. Setting low starting factor produces more initial traffic but also facilitates faster node clustering.

Balance initiating node will first use *family links* to send a message and, if all of them fail, start to use *other links* collection. Messages are sent one by one and targeted ultra node id is stored in the cache that is being emptied once all links have been exhausted. If an ultra node that received *BalanceRequest* message (marked with 1 on Fig. 10) doesn't have enough leaves, it forwards the request (message 2 on Fig. 10) using random *family link* (bloom filter is used). If a message finally reached ultra node that qualifies as balancing source, a *BalanceReply* message is sent (massage 3 on Fig. 10) to the requester with a list of nodes that can be reassigned. Balancing source doesn't remove leaves from its collection yet, because it is not certain how will requester node proceed (it is possible that, due to latency, requester node has more than one balance request active and will not be in position to reassign all of the leaves offered, or requester may have disconnected in the meanwhile, etc.).

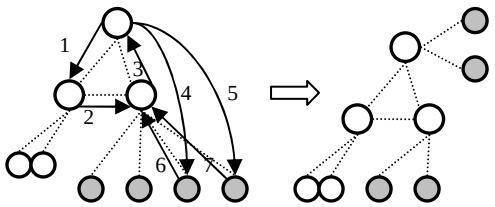

Fig. 10 Ultra node load balancing process

When the requester node receives the balance reply message it sends *JoinMe* messages to proposed leaves (messages 4 and 5 on Fig. 10). Upon receiving *JoinMe* messages leaf nodes update their ultra node link and send *LeafLeft* message (messages 6 and 7 on Fig. 10) to the old ultra node. When an ultra node receives *LeafLeft* message, it deletes the respective link from its leaf links collection.

4.2.5 Voting process – forming family links

Ultra node forms its family links based on the votes it receives from its leaves. Leaves keep track of number of results they've received from other ultra nodes' leaves. Leafs periodically compile a list of N ultra nodes that

have been responsible for the majority of results and, if the number of results is bigger than defined minimum size (to reduce the traffic), send their votes via *VoteMessage* to their ultra node. Upon receiving their leaves' votes ultra nodes are updating their routing tables according to the algorithm in Table 2.

```
updateRoutingTable(voteMsg)
  update new swap list with votes from voteMsg
  sumSwap = sum of all votes from new swap list
  sumFmly = sum of all votes from current (family) list
  IF       (voteCounter>VOTE_CNT_THRESHOLD )
    AND (sumSwap>=MERGE_TRSHLD*sumFmly)
   IF (family list is not empty)
    IF (old family list is not empty)
     subtract the old family list from the family
    END IF
    old family list = family
   END IF
   add new swap list into family list
   clear new swap list
  END IF
```

Table 2 Routing table update algorithm

Since *family list* is a finite sorted list (of ultra node ids and numbers of results) it would be wrong to add to that list every time a vote arrives – that would make it impossible for a new ultra node to enter the list because their votes couldn't compare to the votes other nodes have gathered throughout the entire "voting history". To ensure fairness we keep track of three lists: current family list, old family list (a snapshot of previous family list) and new swap list that is accumulated until it is ready to merge into the curent family list. This way, old votes gradually leave the list and new ones have a chance of entering. When ultra nodes update their *family list* they take into account one extra rule: every leaf has to have at least one "representative" in the family list. That is, all ultra nodes that are on top of leaves's list are included in the *family list*. Remaining positions in the list are populated by sorting ultra nodes (result counts) in descending order. This extra rule is introduced to improve search efficiency in those nodes that are, hopefully temporarily, outside their communities. This way, their ultra node has a direct link to one of the ultra nodes of the desired community. In opposition, if this rule wasn't employed, such leaf nodes would get outvoted and lose short path route to their communities.

Aside from the voting process, hybrid network's ultra nodes subnet behaves analogous to the pure network in a sense that it maintains *family links* collection and *other links* collection and routes queries according to the same principle. Fig. 11 shows a hybrid network consisting of 3 semantically very distant communities with 2 family links, 3 other links and 10 leafs per ultra node after 100 iterations (100 queries by each node).

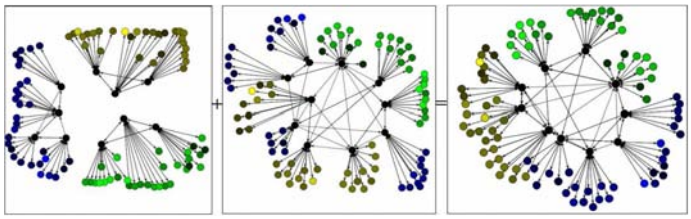

Fig.11 *family links* + *other links* = all links (100 nodes)

If a leaf node fails (disconnects) in hybrid network, its ultra node simply discards the corresponding link. If an ultra node disconnects its leaves repeat the bootstrapping process and (since ultra nodes from their semantic community are probably on top of their list) hopefully find their new ultra nodes within the same community.

4.2.6 Membership changes

When a node joins hybrid network it performs the same steps as when bootstrapping. The only prerequisite is that the joining node "knows" some existing node (we use randomly selected node in our simulations). Departure of leaf nodes doesn't present a problem for other nodes – if they leave they inform their ultra node of their departure and if they fail ultra node detects it eventually (when it tries to send a message to the leaf node) and removes the link from the collection. The same can't be said for the ultra nodes. When an ultra node leaves the network all its leaf nodes have to find another ultra node. Leaves attempt to join ultra nodes in the order of response counts they've provided in the past which improves their chances to remain within the right community. It is better when ultra nodes leave and inform their leaf nodes because then leaf nodes will immediately try to join other ultra peer. Otherwise, a leaf node will detect that it's ultra node is dead when it tries to send a query message to it. In that case, that query will be recorded as failed query and user will have to wait until it reconnects to some other ultra node. Fig. 12 shows the network properties when approximately 60% of nodes fail or leave in a short period. Failing (leaving) nodes are picked at random - therefore both ultra and leaf nodes are failing at random. Top chart in Fig. 12 shows net size with separately marked ultra node count and leaf node count (and their sum – net size). It is evident that, unlike pure network, failing ultra nodes are degrading recall considerably more than leaving ultra nodes. That is why, within hybrid protocol, we differentiate these two events and retain *UltraNodeMessage* in the protocol. In both cases, recall improves after node departures stop and returns to its original level.

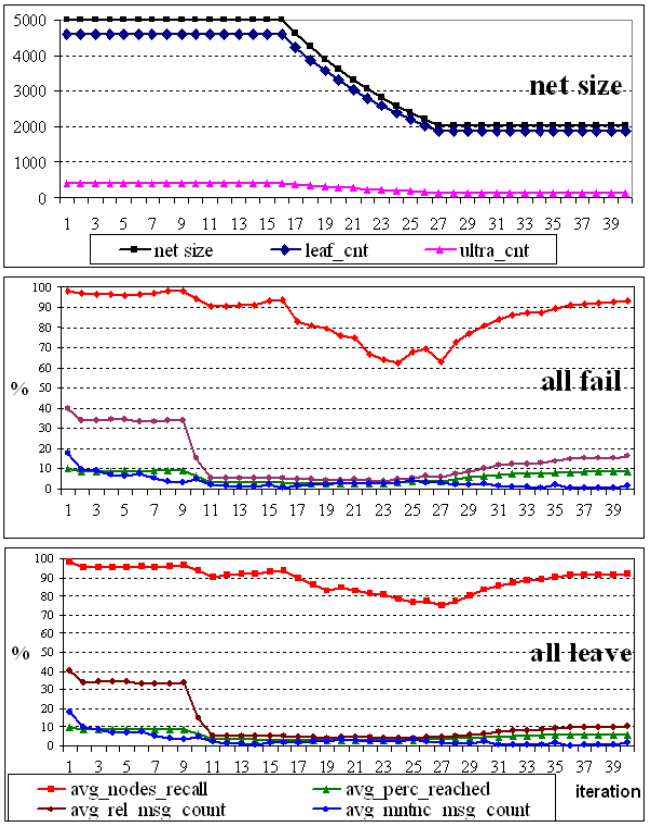

Figure 12. Effects of 60% nodes failing/leaving the hybrid network

5 Experimental results

The experiments were conducted using PlanetSim – an overlay network simulation framework [11]. Within the framework we've implemented Gnutella, Pure and Hybrid protocol. Data set consists of 982 articles from six categories: cooking recipes, Greek mythology, basketball players' biographies, musicians' biographies and programming code (SQL and C code gathered through an application for testing students). Retrieval results are compared to the results that would be retrieved for the same query in a centralized environment, i.e. for a given query we compute the recall as:

$$recall := \frac{no.\ of\ docs\ retreived\ in\ P2P\ env.}{no.of\ docs\ retreived\ in\ centralized\ env.} \qquad (6)$$

meaning that, every time a new query is created during a simulation, all active peers are polled in iterative fashion and the number of documents that match the query is stored so that it can be compared to result set retrieved using P2P search. Our simulations are currently restrained to maximum 10^4 nodes due to our hardware limitations. Fig. 13 shows average recall comparison in static conditions when net size varies from 10^2 to 10^4 nodes. Gnutella 7/17 means that each node maintains 17 links and forwards the query to 7 randomly chosen nodes. Pure 5/5/5 means that each node in Pure network maintains 5 *family* and 5 *others* connections and forwards the query to maximum 5 nodes. Hybrid 5/5/5 1:15 means that every ultra node in hybrid network maintains 5 *family* and 5 *others* connections and forwards the query to maximum 5 ultra nodes and that every node has maximum 15 leaves while every leaf is connected to (maximum) 1 ultra node.

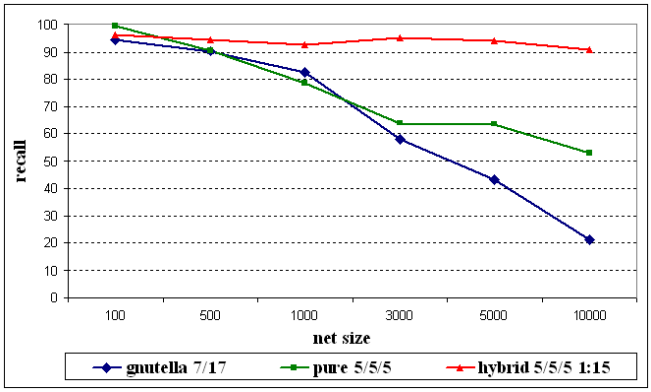

Fig 13. Average recall comparison

As shown, both Pure and Hybrid protocol outperform Gnutella protocol as the size of the network increases while producing significantly less messages (Fig. 14 – note the log scale). Pure network's moderate results can be explained with small number of known nodes (5+5, see Fig. 16).

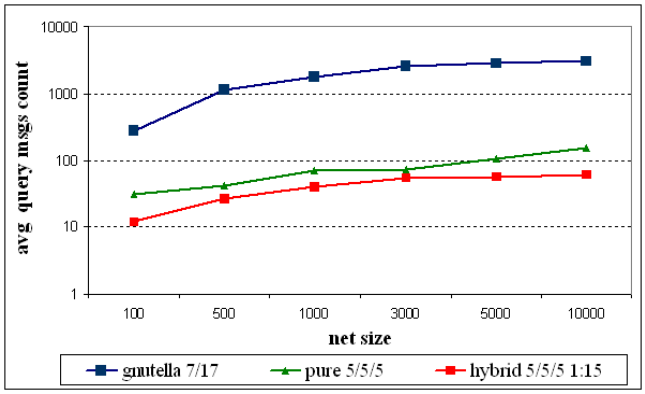

Fig 14. Average query message count

Fig. 15 shows the average message count in more detail, when queries are broken down into categories according to the number of documents that match the query.

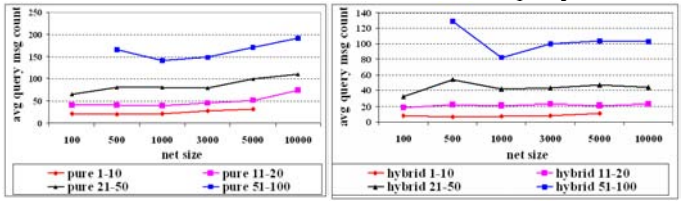

Fig.15 average message count by relevant documents count categories

Notice that query message count is practically constant, especially for the hybrid protocol.

Fig. 16 shows influence of *family* and *others* collection sizes on the recall in both networks. Collection size

shows influence only on pure network at these network sizes (below 10^4). This is logical because ratio of query routing nodes between these two networks is 1:15.

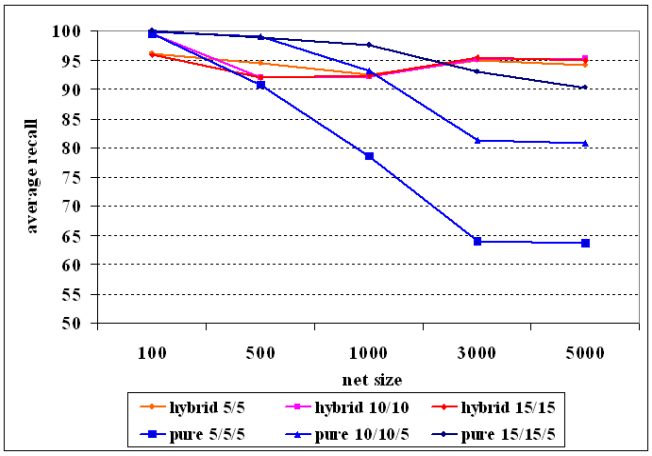

Fig. 16 Influence of *family* and *others* collection sizes on the recall

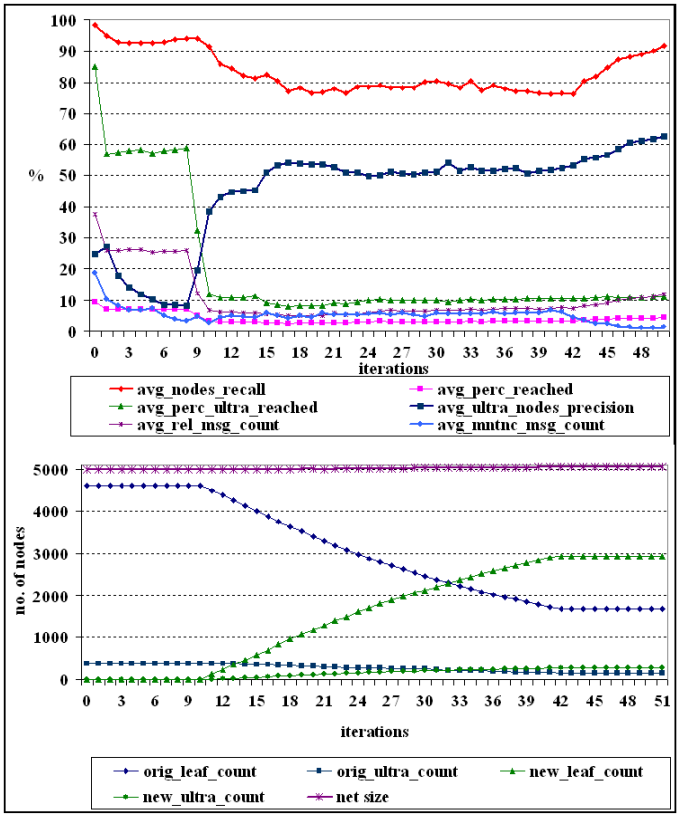

Fig. 17 Hybrid network 15/15/5 1:15 in a very dynamic environment

Finally, Fig. 17 shows hybrid (15/15/5 1:15) network properties in a very dynamic environment: node connections for 5000 nodes are formed during first ten iterations and then, between iterations 10 and 40, approximately 3000 new nodes join, 1500 leave and 1500 fail. For instance, at iteration 31, there are already more new nodes (both ultra and leaf nodes) than original nodes – the ones used to setup the network. It is apparent that, during such rapid membership changes, recall drops (to approximately 80%) and maintenance messages count increases (as the new nodes position themselves within the network). Lower recall rate is caused by the new, unsettled nodes. This is apparent because, as soon as the membership changes stop, recall and precision increase since new nodes have positioned themselves and there are no additional new ones to diminish the recall. Precision is here defined as:

$$nodes\ precision := \frac{no.\ of\ nodes\ that\ should\ have\ been\ reached}{no.\ of\ nodes\ reached} \quad (7)$$

5 Conclusion

Content-based search is a challenging problem in P2P environments. Many researches have proposed structured P2P networks that organize routing structures according to the underlying semantic space and use some kind of distributed hash table functionality to achieve fast (and scalable) retrieval. We focus our attention on situations when there is no prior knowledge of the semantics of data. To that purpose, a set of messages and algorithms for pure and hybrid self-organizing P2P network has been proposed. In the proposed network, nodes organize themselves according to the semantics of data they share: similar nodes are clustered into semantic communities. This way most queries will not have to travel outside their communities. We find pure network simpler, more robust and less susceptible to obstruction while hybrid network reduces query network traffic and therefore scales better. Experiments show that pure network handles node failures slightly better than hybrid network, but we find hybrid superior in other aspects. Most notably, hybrid network achieves better recall than pure network with roughly 2 or 3 times less messages. Both networks are simple and robust which is important in highly transient P2P environments. Proposed networks cannot guarantee logarithmic scaling of query resolving messages typical for the structured P2P systems. Instead, they generate traffic that depends on sought community size (and query similarity threshold) and are actually independent of other communities' sizes. For instance, in our experiments hybrid network consisting of 5000 nodes achieves 94% recall with average 16,24 messages that reach approximately 0,33% of the network. When analyzing in detail, we find that recall is mainly diminished by nodes that are searching for rare content (e.g. there are only 1 or 2 matching documents in the whole network). Such queries often fail because there are not enough nodes in the network to form a community. We expect that nodes with such distinct semantics will always present a problem due to the very nature of P2P systems where "strength lies in numbers".

Future work includes tuning of the system's performance as well as dealing with some real world aspects of the protocol like malicious peer activities and network obstruction. Ultimately, we hope to make network available to general population.

References:

[1] Berry M., Drmac Z., Jessup E., Matrices, vector spaces, and information retrieval. SIAM Review, Vol. 41 No. 2, 1999, pp. 335–362.

[2] Deerwester S., Dumas S., Furnas G., Landauer T., Harsman R., Indexing by latent semantic analysis, J. American Society for Information Science, Vol. 41, 1990, pp. 391-407.

[3] Ratnasamy S., Francis P., Handley M., Karp R., Shenker S., A scalable content-addressable network, In Proceedings of the ACM SIGCOMM, 2001, pp. 161-172.

[4] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., Balakrishnan, H., Chord: A scalable peer-to-peer lookup service for internet applications, In Proceedings of the ACM SIGCOMM, 2001

[5] Zhao B. Y., Kubiatowicz J., Joseph A., Tapestry: An infrastructure for fault-tolerant wide-area location and routing, Technical Report UCB/CSD-01-1141, 2001.

[6] Rowstron A., Druschel P., Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In Proceedings: IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), 2001, pp. 329-350.

[7] Tang C., Xu Z., Dwarkadas S., Peer-to-peer information retrieval using self-organizing semantic overlay networks, In Proceedings of ACM SIGCOMM, 2003, pp. 175–186.

[8] Li M., Lee W., Sivasubramaniam A., Semantic small world: An overlay network for peer-to-peer search, In Proceedings of the 12th IEEE International Conference on Network Protocols, 2004, pp. 228-238.

[9] Tang C., Dwarkadas S., Xu Z., On scaling latent semantic indexing for large peer-to-peer systems, In Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval, 2004, pp. 112-121.

[10] Gravano L., Molina H.G., Tomasic A., GlOSS: text-source discovery over the Internet. ACM Transactions on Database Systems, Vol. 24, No. 2, 1999, pp. 229-264.

[11] García P., Pairot C., Mondéjar R., Pujol J., Tejedor H., Rallo R., PlanetSim: A New Overlay Network Simulation Framework, Software Engineering and Middleware, 2005, pp. 123-136.

[12] Bloom B., Space/Time Trade-offs in Hash Coding with Allowable Errors, Communications of the ACM, Vol.13, No.7, 1970, pp. 422-426

[13] Chawathe Y., Ratnasamy S., Breslau L., Lanham N., Shenker S., Making Gnutella-like P2P Systems Scalable. In Proceedings of ACM SIGCOMM, 2003, pp. 407-418.

[14] Kalogeraki V., Gunopulos D., Zeinalipour-Yazti D., A Local Search Mechanism for Peer-to-Peer Networks, In Proceedings of the eleventh international conference on Information and knowledge management. ACM Press, 2002, pp. 300-307.

[15] Lv C., Cao P., Cohen E., Li K., Shenker S., Search and Replication in Unstructured Peer-to-Peer Networks, In Proceedings of ICS 2002, ACM Press, 2002, pp. 84-95.

[16] Yang B., Garcia-Molina H., Improving Search in Peer-to-Peer Networks, In ICDCS, 2002., pp. 5-14.

[17] Tsoumakos D., Roussopoulos N., Adaptive Probabilistic Search for Peer-to-Peer Networks, In 3rd IEEE Intl Conference on P2P Computing, 2003., pp. 102-109.

[18] Crespo A., Garcia-Molina H., Routing Indices for Peer-to-Peer Systems. In ICDCS, 2002.

[19] Cuenca-Acuna F.M., Peery C., Martin R.P., Nguyen T.D., PlanetP: using gossiping to build content addressable peer-to-peer information sharing communities, In Proceeding of the 12th International IEEE Symposium on HPDC, 2003., pp. 236-246.

[20] Tsoumakos D., Roussopoulos N., Analysis and comparison of P2P search methods, In Proceedings of the 1st International Conference on Scalable Information Systems. vol. 152 of ACM International Conference Proceeding Series, ACM Press, 2006.