Rich and Scalable Peer-to-Peer Search with SHARK

Jan Mischke[1] and Burkhard Stiller[2,1]

[1] *Computer Engineering and Networks Laboratory TIK, ETH Zurich, Switzerland*
[2] *Information Systems Laboratory IIS, University of Federal Armed Forces Munich, Germany*

E-Mail: [mischke|stiller]@tik.ee.ethz.ch

Abstract

SHARK is a novel concept and middleware service for search in peer-to-peer (P2P) networks. Rather than flooding a network like Gnutella or imposing numerical IDs on objects like distributed hash tables, it is based on directed routing of keywords in a multidimensional redundant meta-data hierarchy. SHARK arranges nodes and objects in the network and in semantic clusters. In spite of its rich keyword search capabilities, it achieves a high degree of scalability, outperforming random networks by several orders of magnitude. It can easily be adopted for applications as diverse as filesharing, P2P trading, or distributed expert and knowledge market places.

1 Introduction

P2P networks are proliferating rapidly in areas like P2P computing, collaboration, trading, network testing, but maybe on the largest scale in file-sharing applications like Gnutella, KaZaA, or eDonkey.

However, the problem of service or object discovery on remote peers has, as of now, not been satisfactorily solved. While web search engines like Google demonstrate the power and ease of centralized search, it cannot always be adopted for P2P systems. On the one hand, legal issues led to the shut-down of Napster's central search server. On the other hand, avoiding the need for central infrastructure and, hence, the associated investments, administration costs, and legal as well as social implications like censorship, is one of the main driving forces behind the current P2P idea.

Entirely random networks like Gnutella are inherently not scalable: they conceptually rely on flooding the entire network with messages to find objects, *i.e.*, contacting all or a major part of all nodes and asking for the desired information.

More sophisticated approaches have been devised based on the concept of structured overlay networks and structured lookup tables, called distributed hash tables (DHTs). Unfortunately, current DHTs rely on unique, usually numeric identifiers and are thus limited to pure lookup of these identifiers. Search based on keywords remains impossible. Usually, a user will want to specify what she is looking for in terms of keywords. Rich search allows the user to describe keywords and meta-data properties of the objects she is looking for. Furthermore, range searches for an entire class or range of objects within certain limits are possible. For instance, a user could look for a certain genre of music in music filesharing or for all medications for a specific disease in a medical expert system. Additionally, multiple dimensions of meta-data are highly desirable; in the example, the user might not only want to specify a music genre, but also a certain time of release as a second dimension to narrow down the search.

SHARK (Symmetric Hierarchy Adaption for Routing of Keywords) presents a scalable solution to rich P2P search. It is based upon structuring the overlay network and the search space into a multidimensional symmetric redundant hierarchy. Yet it fully supports rich search queries, *i.e.*, search based on meta-data of multiple dimensions and varying granularity as well as range searches. SHARK is primarily targeted at file-sharing applications but can as well be used for all other P2P service and object discovery problems, *e.g.*, in P2P trading and collaboration.

This paper presents related work and derives the SHARK concept in Section 2 and describes the SHARK system, topology, and algorithm in Section 3. It evaluates the solution in Section 4 before it concludes with Section 5.

2 Background and related work

A complete design space and methodology to design distributed search systems has been presented in [10] and led to the development of SHARK as one of the few remaining entirely novel approaches to P2P search. Table 1 shows those parts and example systems that are relevant for P2P.

It is important to separate functional and structural aspects of P2P lookup and search systems. In the functional dimension, there are three functionalities a lookup/search system can provide. First, there are pure keyword search systems mapping keywords onto unique names. Examples include web search engines, mapping keywords onto URLs, and, to a lesser extent, INS/TWINE [2], mapping keywords onto hash IDs that may or may not exist in the network. Second, there are lookup or name routing mecha-

Table 1. Survey of distributed and P2P search in the design space

Design						Keyword Search	Lookup/Name Routing	Keyword Lookup/Routing
Computational						INS/Twine[a]	n/a	n/a
Table	*Central*					(Google)[b]	(Load balancing hub)[b]	Napster
	Complete on Each Node					Groove[a]	(NIC's Hosts.txt)[b]	n/a
	Distributed Table	*Aligned Table Structure and Topology*	*Hierarchical*		*Classical*	n/a	(DNS)[b]	TerraDir, Mutant Query Plans
					Symmetric Redundant	n/a	Pastry, Tapestry, AGILE, Kademlia/Overnet	**SHARK**
			Non-Hierarchical Ordered Space			n/a	CAN, Chord	n/a
		Unaligned Table Structure and Topology			*Table Structured*	n/a	(TRIAD/NBRP)[b]	n/a
					Table Unstruct., Topology Structured	n/a	HyperCuP[a]; Supernode networks: FastTrack (Morpheus, KaZaA, Grokster), Gnutella (Bear-Share Defender, Clip2 Reflector, LimeWire[c]), eDonkey	LimeWire[c], SIL
		Random Table Structure and Random Topology	*No Neighborhood Information*			n/a	Gnutella, Expanding Ring, Random Walk, Associative Overlays	Random Walk, Expanding Ring, LimeWire[c], Interest shortcuts
			Neighborh. Information		*Without Recursion*	(Manual http-Browsing)[b]	Freenet	Local indices
					With Recursion	n/a	Variants of Bloom filters	Bloom filters, e.g. LimeWire[c]
Hybrid Systems						n/a	Yappers, Brocade	(SHARK)

a. Only partial fit into category

b. Not deployed for P2P

c. LimeWire proposes multiple add-ons to Gnutella and is subsequently listed multiple times in the table.

nisms that take unique names as an input and lookup or route towards the corresponding objects like Chord [1]. Finally, there are keyword lookup/routing schemes that combine the first two systems and route towards objects directly based on keywords. SHARK takes this approach.

In the structural dimension, there are several fundamentally different approaches to perform either of the three mappings used for keyword search, lookup/name routing, or keyword lookup/routing, respectively. Search systems can try a computation, apply central tables, replicate all information necessary for search on all nodes, choose a distributed table, or a hybrid approach combining several of the above at different stages. Distributed tables can exhibit a clear structure aligned to the overlay network topology as for the DHTs, be completely randomly distributed, or rely on some structure for either the overlay or the table information without aligning them.

2.1 Functionally related systems

Several other systems applying a combined keyword lookup/routing approach are contrasted to SHARK in the following.

Napster applies a central server and is thus not well suited for pure P2P networks.

Random network approaches like Gnutella, expanding ring search [21], or random walk [7] show severe scalability limitation as they contact all or a major part of all nodes to retrieve information.

Several attempts have been made to address this issue. Interest-based shortcuts [19] can be created in arbitrary topologies based on past successful responses to exploit interest locality and support semantic clustering. Similarly, guide rules are proposed in [3] to create associative overlays and limit flooding to peers who have at least one item in common with the requestor. Peers maintain indices of their direct neighbors's files in [21]. This can effectively spare the last hop in a flooding strategy and, hence, result in better bandwidth efficiency. Various variants of Bloom filters have been proposed to aggregate and compress information on resources in the direction of each neighbor to improve routing efficiency [13], [17], [4]. Unfortunately, none of these systems can scale as well as structured networks, assuming a sufficiently stable environment so that the overhead for managing a structured overlay remains within bounds.

Perhaps most closely related to SHARK is TerraDir [18], which builds a classic keyword tree. However, it appears impossible to define the complete ontology of object descriptions down to the last leaves that TerraDir assumes. Finally, in [12], a multi-dimensional categorization hierarchy is managed by category servers and queries are processed by a hierarchy of meta index servers, index servers, and base servers. For both approaches, their hierarchical nature and, hence, different roles of nodes, make them less well-suited for pure P2P applications.

2.2 Structurally related systems

AGILE [9] describes the concept of a symmetric redundant hierarchy with groups of interest like SHARK, even though only one-dimensional in the AGILE case. However, its hash-based intermediate hierarchy levels limit it to lookup only.

Even though not described this way by the respective authors, Pastry [5], Tapestry [22], and Kademlia [8] essentially build one-dimensional symmetric redundant lookup hierarchies. In Pastry [5] and Tapestry [22], content names and IP addresses of nodes are hashed onto the same numerical identifier (ID) space; this allows name routing when making that node responsible for holding a resource or a link to it that is closest to the resource in the ID space. The hierarchy is created through a digit representation of the ID to a base value and an association of each digit with one hierarchy level, starting from the last (Tapestry) or the first digit (Pastry), respectively. Kademlia [8], commercially deployed in Overnet, follows the same basic approach but uses a bit- (rather than digit-) representation of IDs to enable prefix matching via XORing bit strings.

Chord [1] hashes resource names and node IP addresses to a 128-bit ID. The IDs are arranged in a circle with fingers, with the predecessor node of a resource ID being responsible for providing the resource or a link to it. In CAN (Content Addressable Network, [14]), hashing is similarly applied to map resource names onto an ID in a d-dimensional torus. Nodes distribute responsibility for the ID space among themselves and maintain virtual links to all direct neighbors in the torus. Queries for a name, *i.e.* ID, can then at each node be routed into the optimum direction.

As the symmetric redundant hierarchies presented above, Chord and CAN are highly scalable, but rely on numerical identifiers that restrict their use to lookup only.

For scalability reasons, SHARK also builds a structured topology, but departs from the notion of numerical IDs. It constructs a multidimensional symmetric redundant hierarchy of meta-data at the top level, while relying on random networks at the bottom level. This avoids an over-structuring of the network in light of the frequent changes typically occurring in P2P networks.

3 Searching with SHARK

This section describes the construction and operation of SHARK.

3.1 Scenario and definitions

A peer-to-peer network consists of a set of *nodes* $SN = \{N_i | i = 1..n\}$ connected via a set of *links* $L = \{l_{ij}\}$ (cf. Figure 1). We call $SN_i = \{N_j | \exists\, l_{ij}\}$ the

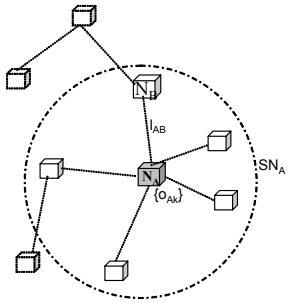

Figure 1. P2P network and definitions

neighborhood of node N_i. Each node $N_i \in SN$ stores a set of *objects* $O_i = \{o_{ik}\}$ that constitute to all the unique objects in the entire peer-to-peer network $O = \bigcup_{i \le n} O_i =: \{o_j\}$, where $m = |O| \le \sum_{i=1}^{n} |O_i|$ due to replication of identical objects onto several nodes. An object is described through *meta-data* $M_j = M(O_j)$.

3.2 Meta-data structure

We impose a *structure* on the meta-data that is essential for the construction and operation of SHARK. We require meta-data to be hierarchical so that $M_j = (M_j^1, M_j^2, .., M_j^l; M_j^0)$. l is the number of levels in the hierarchy, M_j^1 yields a first-level categorisation of the object while $M_j^2, .., M_j^l$ add finer granularity to the categorisation. Finally, M_j^0 is a string expression that further specifies the object within the lowest-level category. We allow multiple dimensions for the meta-data hierarchy,

$$M_j^1 = (M_j^{11}, M_j^{12}, .., M_j^{1d_1})$$

$$M_j^2 = (M_j^{21}, M_j^{22}, .., M_j^{2d_2(M_j^1)})$$

…

where d_1 denotes the number of dimensions on level 1, $d_2(M_j^1)$, the number of dimensions on level 2, and so on.

Note that the dimensionality on each level i can in the most general case be different depending on the higher level category M_j^{i-1} chosen.

Figure 2 illustrates this meta-data structure. As music file sharing is currently the most popular application for P2P search, we have based the example on music categorisation from allmusic.com. M^{11} yields the top-level music genre, that is further divided into subgenres M^{21}. In the second dimension M^{12}, music is classified by decade of release, then by more granular timing M^{22}. M^0 is the search string, e.g., 'John Patton: Let 'em roll'. As stated above, applications may choose to add dimensionality just for certain categories, *e.g.*, add an 'instrumentation' dimension to 'rock&roll' in addition to subgenres and timing.

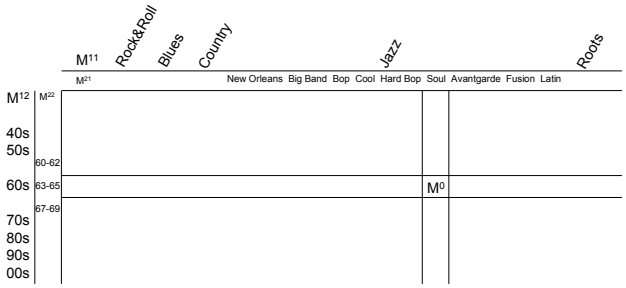

Figure 2. Multidimensional Meta-Data Hierarchy

Other applications can simply develop different schemes for different scenarios like categorisations of objects in medicine or jurisdiction. For multi-purpose networks, a higher level can easily be added to first distinguish, in the example, among music, medicine, and jurisdiction. For illustration purposes, we restrict ourselves in the following to two dimensions and levels without loss in generality; further levels and dimensions can be added at the discretion of the application developer.

3.3 Overlay structure

SHARK arranges nodes into a multidimensional *symmetric redundant hierarchy* (SRH). The overlay topology exactly matches the structure of the query meta-data such as to exploit the alignment for query routing. For ease of presentation, we restrict the descriptions in this section to two levels and two dimensions. Figure 3 illustrates the topology.

Each node is assigned to a *group-of-interest* (GoI) according to the objects it stores and to its prior request behaviour (cf. Section 3.5.2). Each GoI represents a leaf in the hierarchy. Let $P^{(1)} = (p^{11}, p^{12})$ denote a position on level one of the hierarchy, $P^{(2)} = (p^{11}, p^{12}, p^{21}, p^{22})$ a position on level 2. The values p^{ij} numerically represent the respective meta-data information m^{ij}. Node A in the figure

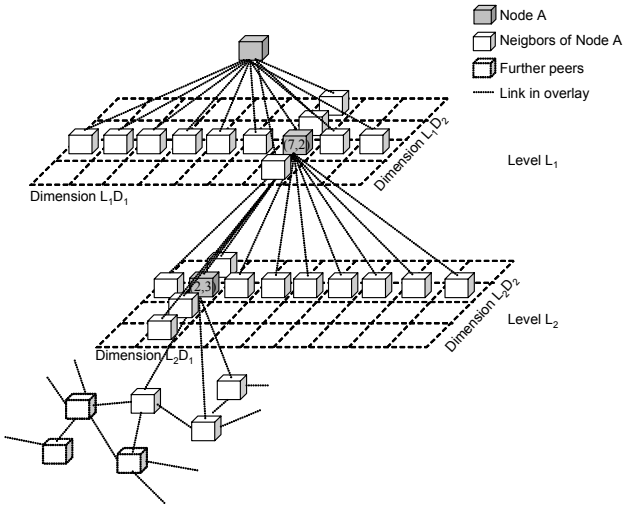

Figure 3. Multidimensional Symmetric Redundant Hierarchy in SHARK

would then be a member of the GoI on leaf position $P_A = (7, 2, 2, 3)$. In contrast to a classic hierarchy, an SRH adds redundancy so that all peers have symmetric roles in the overlay; *i.e.*, each peer can assume the role of the root of the network or be on any other level. This improves fault tolerance and load balancing as there is no single node acting as a root, and waives the necessity of central infrastructure, hence removing the largest roadblocks for an adoption of hierarchical structures in P2P networks.

SHARK adds redundancy as follows. Each node N_A on a leaf position $P_A^{(2)} = (p_A^{11}, p_A^{12}, p_A^{21}, p_A^{22})$ also assumes partial responsibility for the parent position $P_A^{(1)} = (p_A^{11}, p_A^{12})$, the parent's parent, and so on up to the root (in the two level case, the parent's parent is identical to the root). Hence, each node is virtually replicated on every level of the hierarchy (cf. dark grey nodes in Figure 3).

The partiality of the responsibility results from two reasons. First, many different peers share the same parent position, thus inherently distributing the load of that position among themselves. Second, a node only maintains links to a subset of the positions on the respective next lower level in the hierarchy. As indicated in the figure, those positions form the relevant level-i-subset for a node N_A that differ from position $P_A^{(i)}$ in only one dimension. We have chosen this approach to limit state information on the nodes as well as the maintenance burden when nodes join or leave the network, thus increasing scalability of the system at the cost of only one additional hop for query routing per level (cf. Section 3.4). Let N(P) denote a node on position P and b_{ij} the number of different positions on

level i in dimension j. Then an arbitrary node N_A maintains the following neighborhood within the hierarchy:

$$SN_A^h = \{N(p_A^{11}, p_A^{12}, p_A^{21}, i);N(p_A^{11}, p_A^{12}, j, p_A^{22}); N(p_A^{11}, k);N(l, p_A^{12}) \mid i, j, k, l \in \aleph; i \le b_{22}, j \le b_{21}, k \le b_{12}, l \le b_{11}\}$$

The routing table that N_A maintains is straightforward: it comprises the meta-data descriptions of the positions indicated above (and shown in the figure) and the IP addresses of the respective nodes in SN_A^h as corresponding next hops.

Within the leaf GoIs, peers maintain links to further neighbors SN_A^R, as indicated in the figure. The overlay network at this stage is, however, unstructured or random. It has been shown that such networks exhibit a power-law distribution of links [16].

3.4 Search, query routing, and object insertion

Search for objects in SHARK is based on query routing. A *query* $Q(M_q, t_{struct}, t_{rand})$ is defined through a meta-data description M_q of the desired object(s) and thresholds t_{struct} and t_{rand} for the minimum required similarity of object and query description for the structured and the string expression part of the meta-data description, respectively. SHARK returns a set of *query answers*

$$QA(M_q) \subseteq \left\{ O_{ik} \mid S[M^{jl}(O_{ik}), M_q^{jl}] \ge t_{struct}, \forall(j, l); \; S[M^0(O_{ik}), M_q^0] \ge t_{rand} \right\}$$

where $0 \le S(M_a^{jl}, M_b^{jl}) \le 100\%$ is a *similarity metric*. The development of reasonable similarity metrics is orthogonal to and hence not focus of this work. A most simple approach is an exponential transform $\exp(-D_E)$ of the edit distance D_E (cf. [6], p.300ff, p.83).

With the query meta-data structure and the overlay topology aligned and defined as above, queries can efficiently be routed towards relevant content. When a node N_A initiates or receives a query $Q(M_q,t_{struct},t_{rand})$, it sends or forwards it, respectively, to all neighbors whose position meta-data exhibits a similarity with M_q greater than the threshold t_{struct}. It further adds information (l_c,d_c) on the current level and dimension in the hierarchy that has been resolved to avoid duplication of effort. With a certain *pruning probability* p_p, the requesting or currently forwarding node may already itself be on the correct position for the next hop, so that a 'part of the tree can be pruned off' and some hops can be avoided. The forwarding process is repeated until the query is either aborted due to a lack of suitable categories or until it reaches the corresponding leaf position in the hierarchy. From there on, it is flooded throughout the GoI along the random power-law network. Figure 4 shows the formalized algorithm. Every node that caches a link to an object with sufficiently high similarity (greater than t_{rand}) returns that link to the requestor.

```
function flood(Q); // floods the GoI

function forward(Q,lc,dc) {
    if (dc==d(lc) && lc==l) {
          flood(Q);
          exit,
    }     // last level and dimension reached
    else if (dc==d(lc)) { // last dimension on level
          dc=0;
          lc++;  // set dimension 0 on next level
    }
    dc++;
    For N in SNA^h(lc,dc) // Neighbors on (lc,dc)
          if S[Mq^lc,dc, M(P^lc,dc(N)] >= tstruct {
          // Sufficient similarity with position?
                Send(Q,lc,dc) to N;
          }
    }
    if S[Mq^lc,dc, M(P^lc,dc(NA)] >= tstruct
    // pruning: can move down hierarchy on same node
          forward(Q,lc,dc); // start at top
}
```

Figure 4. Query Routing (Pseudo Code)

The final flooding can be avoided under certain circumstances. If partial results are sufficient, it can be replaced by a number of techniques like random walk or expanding ring search [7]. Alternatively, peers in a GoI can maintain links to all or a major part of all objects corresponding to that GoI. Bloom filters can be used to compress the potentially high amount of state information required [15], [17]. All of these improvements are orthogonal to the SHARK concept and will not be considered in more detail here.

SHARK routes a query to the corresponding GoI. In order for an object to be found, it is required that it has been *inserted* or *published* to that GoI before. When a user wants to make an object available, she assigns a SHARK-conform meta-data description M_{IR} and initiates an insert request $IR(M_{IR},r)$, where r is a replication parameter. The insert request is routed through the hierarchy analogous to the query routing. When it reaches the correct leaf GoI, the contacted node N_C will store a link to the object. Furthermore, it forwards $IR(M_{IR},r'=0)$ to r neighbors to ensure appropriate replication, setting a new $r' = 0$ to avoid recursiveness. If $r > |SN_C^R|$, it forwards to the last neighbor a modified request $IR(M_{IR}, r' = r - |SN_C^R|)$.

Structure and query routing mechanism in SHARK have been designed to inherently allow range queries. Usually, only one category or position will correspond to a given

query. However, if a user wants to search multiple categories at once, she can simply use a disjunctive combination to specify the respective M_q^{jl}, provided the similarity metric has been appropriately chosen to support combinations. Similarly, she can use wildcards like '*' to initiate an incomplete request and search, *e.g.*, for 'Bop Jazz' regardless of time of release. SHARK automatically resolves the query into multiple replicas where required to incorporate range queries. Finally, it is obvious that wildcards and other rich query descriptions can also be used for M_q^0, yielding a part of or even all objects in a certain category.

3.5 Overlay network management

With the SHARK structure and its use for query routing defined, we now describe the creation of the structure through node insertion and its management and adaptation over time through adaptive semantic clustering.

3.5.1 Node insertion. New nodes can easily join the SHARK overlay using essentially the same mechanisms as for query routing. When a node wants to join, it contacts an arbitrary peer N_C, sending a node insertion request `insert_node(`M_{own}`)` with a meta-data description of its own GoI, which is determined as outlined in the next section.

N_C then starts routing the request to the appropriate leaf in the overlay. The joining node copies at each hop in the hierarchy the set of neighbors for the current level from the forwarding peer (N_{FP}), and uses this information for its own links and, thus, routing table. At the same time, N_{FP} adds the new peer to its routing table. If N_{FP} already has two entries for the same position, the most-recently-seen-alive neighbor is kept while the other neighbor is replaced with the joining node.

The 2-redundancy thus created increases SHARK's fault tolerance. When a node fails that would have been the next hop for N_{FP}, the alternative redundant node will be used until the failed node becomes available again or until it is eventually replaced. Even more important for fault tolerance, however, is the redundancy built into the system. When both alternative next-hop peers fail, N_{FP} simply forwards the query to an arbitrary member of its GoI. Remember that all members of a GoI have routing tables that are equivalent in their function but usually differ in terms of the exact neighbors. N_{FP} then additionally requests a routing table update from its GoI neighbor as a replacement for one of the failed nodes in its own routing table. Should under extreme circumstances both mechanisms fail, N_{FP} has to re-establish its links to the overlay network by sending an `insert_node(`M_{own}`)`.

3.5.2 Adaptive semantic clustering. The categorisation strongly relates to the specific application. For instance, an extensive music classification has been proposed in [11]. While the application developer, hence, defines the initial categorisation hierarchy, SHARK provides means for adaptive semantic clustering of nodes.

When a node N_A joins the network, its objects $\{o_{Ak}\}$ determine the initial GoI it is assigned to. The insertion algorithm examines the meta-data descriptions $M(o_{Ak})$, starting at the top level and first dimension $M^{11}(o_{Ak})$. At each level and dimension, the maximum number of objects with identical meta-data determines the position in SHARK for the new node.

This way, SHARK achieves a semantic clustering of nodes into groups of interest that share similar objects. Furthermore, the overlay adapts over time according to nodes' queries. When a node N_A initiates a query Q_{Ai} and receives responses QA_{Ak}, it stores the meta-data of the first object QA_{A1} in a local FIFO queue q_F with an additional absolute maximum time-to-live TTL_{qF} for its elements. The same algorithm used for node insertion also examines the meta-data in q_F. Whenever the number of identical meta-data M^{jl} on any level and dimension exceeds a threshold t_{VN}, N_A inserts an additional *virtual node* N_A' into an arbitrary child GoI of M^{jl}, using the same procedure as for real node insertion. Vice versa, whenever the number of identical meta-data M^{jl} in q_F that lead to the creation of a virtual node falls below a threshold $t_{del}<t_{VN}$, the virtual node is deleted. The use of virtual nodes serves two purposes. First, it allows SHARK to *semantically adapt* over time; nodes move towards the content they like and request. Further queries will be served more *efficiently* (*i.e.*, with a higher pruning probability), as the requestor N_A can initiate a query through its virtual node N_A' directly within the appropriate GoI (or, at least, at a well-suited parent). Second, virtual nodes achieve *fairness* in the P2P system and help cater for *heterogeneous* capabilities of peers. Those peers initiating queries most frequently will eventually also carry the highest network burden as related to search due to the routing and object insertion load on their virtual nodes. The TTL_{qF}, in contrast, serves to avoid virtual node creation for nodes with low query frequency or low uptime.

In addition to the movement of nodes within the SHARK hierarchy, the categorisation itself can also adapt over time. A node N_A can create a new category with an `insert_GoI(M(GoI`$_{new}$`))` request. N_A automatically becomes the first member of GoI_{new}. The routing procedure for `insert_GoI` is identical to the query routing mechanism, except that exact matches at each level are

required (t_{struct}=100%); a resolution into multiple requests is not allowed. After successful creation of GoI_{new}, N_A sends an `announce_GoI(M(GoI_new))` to all sibling GoIs or to the parent GoI if no siblings yet exist. It learned about the siblings and the parent during `insert_GoI` just the same way as nodes learn about neighbors during node insertions (cf. Section 3.5). Within the sibling GoIs (or the parent GoI), the `announce_GoI` is flooded so as to notify all peers of GoI_{new} that may become involved in routing toward it. Potential requestors to GoI_{new} learn about its existence through an extended `announce_GoI_all(M(GoI_new,1),M(GoI_new,2),...)` message. We expect new category creation to happen in local bursts, with peers trying to add segmentation to one area of the multidimensional hierarchy. In order to accumulate several category insertions, a creator N_A waits with the group announcement for a time-out t_{cat}. In the mean time, it caches all `announce_GoI` messages it receives from its siblings (or children). After t_{cat} has expired, it efficiently floods the `announce_GoI_all` message including all cached announcements throughout the SHARK by using wildcards for all levels and dimensions in the query routing mechanism. All other creator nodes whose new GoI has been announced this way clear their caches.

A flooding of the `announce_GoI_all` message can be avoided under certain circumstances or in some applications. If a linear order can be imposed on SHARK categories along all dimensions and on all levels, requestor peers do not need to be notified about a new category insertion. Consider the following example. SHARK is used for a P2P based newspaper article archive. GoIs are built by date of appearance of an article in one dimension and by first letter of the author name in another dimension. GoIs can be searched by title. A total linear order can be imposed on both dimensions: increasing date and lexicographic order in the alphabet, respectively. Say, a user initiates a query Q(M^{11}='06/23/2002',M^{12}= 'Duck',M^{0}='Money',80%,70%). It is irrelevant for the user whether there is only a bin for '2002' or whether there also exists a bin for 06/2002, the request will be routed correctly; the same applies to a bin 'A-F' or a bin 'D', respectively.

3.5.3 Triggered Group-of-Interest splitting. As described above, peers may create GoIs at their discretion at any time, effectively splitting a GoI when introducing several new GoIs one level below an existing one. In order to avoid too large GoIs that limit scalability, SHARK also measures GoI sizes and actively triggers nodes to split large GoIs. Whenever a query Q is flooded throughout a GoI starting at a node N_A, a hop counter HC within Q is increased at every instance of forwarding. When a node receives Q with HC>t_{maxGoI}, a threshold for the maximum reasonable GoI size, it sends a `group_exceeding_limits()` announcement to N_A. N_A counts these announcements; when their number exceeds a second threshold $t_{min\_affected}$, it floods a request `split_group()` through the GoI to split the group. Nodes thus obtain the information required to know when GoI-splitting is sensible. Once a `split_group()` message has been sent, all nodes within that GoI stop counting hops and sending `group_exceeding_limits()` announcements to avoid flooding N_A at subsequent request queries in case the GoI was not split in the mean time.

Rather than relying on users to split groups into newly defined GoIs, SHARK could itself actively perform this action. However, as SHARK cannot invent new semantic sub-categories, this would require that a total linear order can be imposed on existing category descriptions (cf. above).

4 Evaluation

In this section, we evaluate SHARK with respect to five crucial aspects.

4.1 Aggregate number of messages for a query

The number of messages sent to resolve a query constitutes the most critical issue of a scalable P2P network; it determines both, the aggregate bandwidth consumed and the aggregate processing load on peers.

The average number of messages for a query depends on the distribution of nodes and queries to GoIs. We have modeled two distributions: a uniform distribution (most simple case) and a bimodal Zipf distribution. It has been observed (cf., *e.g.*, [6]) that document popularity follows a Zipf distribution, and it is reasonable to make the same assumption for GoI popularity in terms of the number of member nodes and the relative request frequency. A Zipf distribution ranks objects by popularity and yields the probability that a request is made for an object with a certain rank i as $p(Q_i) \sim 1/i^{\alpha}$. More recent studies [20] of P2P networks like Gnutella suggest a bi-modal Zipf distribution with constant (α_1=0) popularity up to an inflection point. We choose α_2=1, which corresponds to the original version in [6] and is the average reported in [20], and we set inflection at t_{inf}=1% of possible GoI ranks. The number of messages for a query is $n_m = n_{h,h} + n(GoI)$, where $n_{h,h}$ is the number of hops in the hierarchy and n(GoI) is the number of nodes in the GoI contacted, as a message is passed to any node in the GoI due to flooding[1]. With the

pruning probability p_p as in Section 3.4 and with the definition of n_{step} as the maximum number of total steps in the hierarchy, *i.e.*, the number of levels times the number of dimensions on each level (if constant), $n_{h,h}$ becomes

$$n_{h,h} = n_{step}(1-p_p)^{n_{step}} + \sum_{i=1}^{n_{step}-1} i p_p (1-p_p)^i$$

Figure 5 compares different approaches, assuming a pruning probability p_p equal to the relative frequency of the most popular category, and a hierarchy with two levels and two dimensions per level. With a reasonably sized hierarchy of 256 GoIs, SHARK shows an improvement of two orders of magnitude over Gnutella. As expected, the overlay performs better when queries are evenly distributed than in the Zipf case, except for very small networks due to pruning effects. In both cases, the number of messages remains almost constant while the GoIs are being 'filled', the increase becoming roughly linear afterwards. SHARK exhibits logarithmic scaling properties when GoI splitting is active. The lowest solid line is obtained when a GoI splits upon exceeding a size of 100 nodes; for simplification, we assume one level to be added with four (split along both dimensions) equally sized groups at each event of splitting.

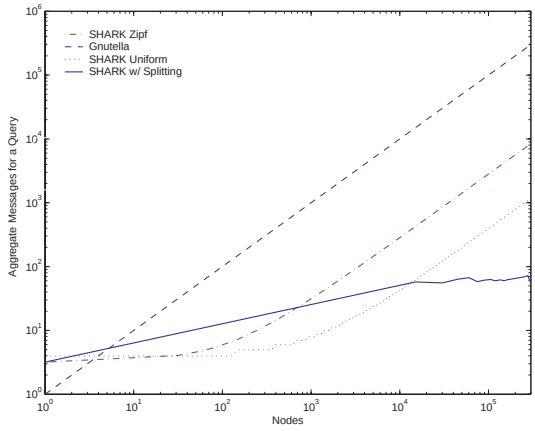

Figure 5. Messages: Comparison of approaches

4.2 Cost of node and object insertion

The cost of a node join is the number of hops in the hierarchy $n_{h,h}$ that a node makes until it reaches its GoI plus two hops for redundancy within a GoI, times two messages per hop: one for the forwarding, one for copying the relevant part of the routing table information of the forwarding node. The cost of each object insertion is also the number of hops $n_{h,h}$ to a GoI plus two messages within a GoI for redundancy, assuming a replication parameter r=2. Both are plotted in Figure 6. As for the remainder of this section

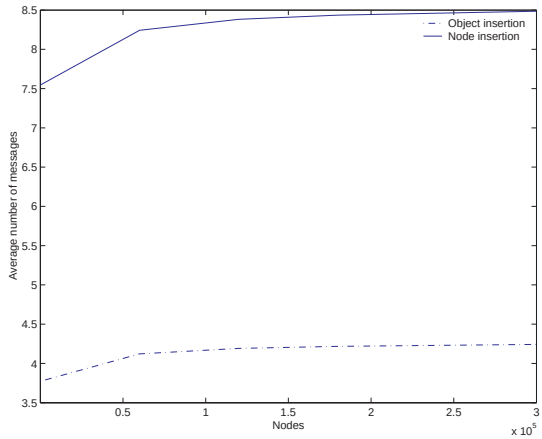

Figure 6. Node and object insertion cost

(unless otherwise stated), we ass‘‘ume a pruning probability p_p=30%, a bimodal Zipf distribution for GoI popularity as described above and initially 256 GoIs with GoI splitting at 100 nodes.

4.3 Number of query hops and latency

We will now look at the average number of hops required to find an object (or multiple objects matching a query) as a proxy for query latency. Figure 7 shows this average for a mean node connectivity of $\dot{I} = 3.4$ [16].

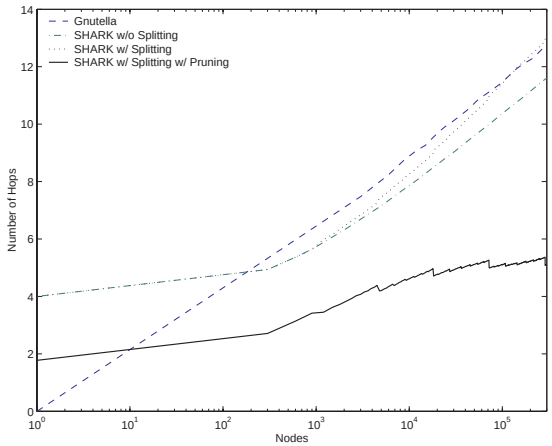

Figure 7. Number of hopS

Gnutella exhibits the expected logarithmic behaviour (dashed line). SHARK shows a similar number of hops as Gnutella when neither GoI splitting nor pruning occur (dash-dotted line), with slightly worse performance for small networks as 4 hops are always required to reach the correct GoI. Few additional hops are needed when GoI splitting occurs but no pruning (dotted line). This is

1. This assumes a loop-free GoI topology. While this will usually not be the case, it does not essentially change the results of the analysis, as we made the same assumption for the Gnutella reference case.

straightforward to understand having in mind that a GoI split in two requires one additional hop, whereas one further hop in a random network yields a factor 2.4>2 of additional contacted nodes. We expect, however, the lower (solid) line to more realistically reflect SHARK's behaviour, assuming GoI splitting at n(GoI)>100 and a pruning probability p_p=30%. In this case, SHARK clearly outperforms Gnutella-type networks for number of hops or latency and exhibits even sub-logarithmic scalability.

4.4 Routing table size

Figure 8 shows the average routing table size for a SHARK node. It initially remains constant at roughly 35 entries, and evolves logarithmically once GoI splitting begins, reaching a size of 61 entries in a network of 300K nodes.

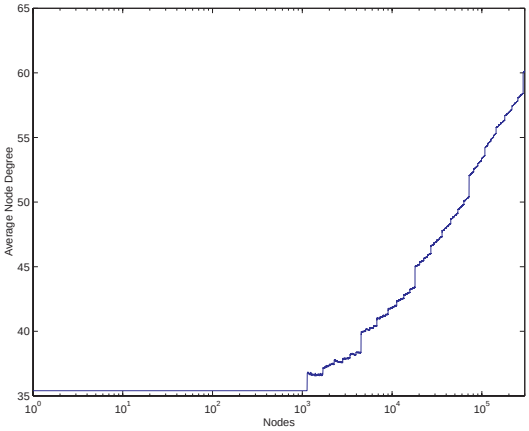

Figure 8. Routing table size in SHARK

4.5 Cost of network management and growth

Figure 9 shows the bandwidth cost in terms of the aggregate number of messages sent in SHARK as the network grows from 10 nodes to 300 million nodes. Even at 300 million nodes, the total number of messages received/sent per node per day remains at approximately 25,000. Assuming an average message size of 154 Byte (cf. [21]), this corresponds to roughly 0.4 KBps bandwidth consumption per peer through SHARK. This compares to $6*10^9$ messages or 80 MBps for a Gnutella network of the same size (with a high enough TTL such as to reach all nodes with a query). The initial roughly linear increase in number of SHARK messages is explained through the initial choice of 256 GoIs that 'fill up' before GoI splitting occurs as the network grows larger.

The figure further shows that query messages clearly dominate the total bandwidth cost, and in the graph the corresponding line even coincides with the total number of messages (thick solid line). The cost of node joins and object insertions as the network grows (dotted line) is almost negligible. The cost of GoI splits (dashed line) would become significant at extremely large network sizes. It is, however, dominated by the announcement messages `announce_GoI_all(M(GoI`$_{new,1}$`),M(GoI`$_{new,2}$`),...)` of new GoIs rather than messages for object re-insertion or re-categorization. Therefore, we chose to aggregate all GoI announcements that happen within an area of the hierarchy for a time-out period t_{cat} of 1 hour (cf. Section 3.5.2). Note that the caching delay does not negatively affect query resolution in the mean time: requests that do not take the most recent GoI splits into account are simply sent to all sub-GoIs at the next higher level in the hierarchy.

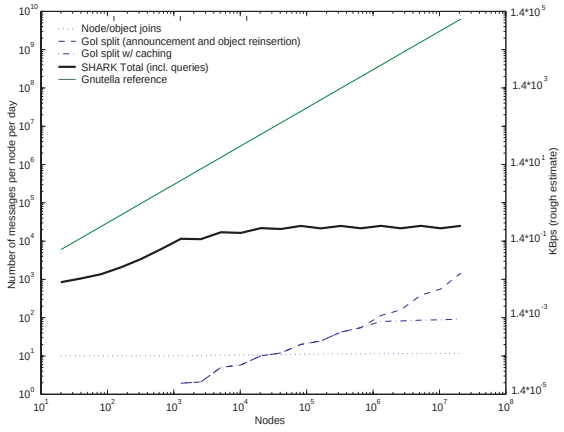

Figure 9. SHARK bandwidth cost in growing network

For the simulation, we made the following calculations and assumptions. The comparison of query messages and network management overhead requires knowledge of growth and query rates. We assumed a very aggressive exponential network growth from 10 peers to 300 million peers in 5 years and a query rate of 17 per peer and hour, in line with measurements in [20]. The network growth determines the number of node joins in any given period of time. We assumed object insertions to occur with the insertion of new nodes, where each node holds an average of 340 objects [21]. The cost of GoI splitting accounts for three types of messages. First, all directly affected nodes (*i.e.*, all nodes in the parent GoI) are notified by an `announce_GoI(M(GoI`$_{new}$`))` message. Second, each affected node transfers the object links it stores and that it is no longer responsible for to the three new sibling GoIs, each with one hop plus two messages to maintain redundancy, and an additional two messages per object to inquire at the object owner about the correct new placement. Finally, the insertion of a new GoI is flooded through the network using `announce_GoI_all`. If the rate of new GoI creations becomes sufficiently large, the caching of announcements for and aggregation within 1 hour before transferring it to the next higher level restricts the number

of network flooding events to a maximum of the number of hierarchy levels per hour.

5 Conclusions

SHARK is a P2P search system based on a combination of a multidimensional symmetric redundant hierarchy and multiple semantic clusters, or GoIs, with randomly connected peers. In contrast to DHTs, it provides rich meta-data search capabilities. In terms of bandwidth scalability, it is expected to outperform Gnutella by several orders of magnitude and, depending on the assumptions for node behaviour, may even achieve logarithmic scaling properties. Even its latency will range well below the already benign logarithmic behaviour of Gnutella. The simulated additional overhead for network management and group splitting in a growing network remains small compared to query traffic.

We expect SHARK to be applied for large-scale filesharing scenarios as well as for a multitude of other scenarios, like sales item discovery in P2P trading, knowledge discovery and expert identification in distributed expert systems and knowledge marketplaces, to name a few.

Going forward, security issues like the effects of malicious node behaviour should be investigated. Furthermore, we will build a filesharing application based on SHARK, particularly looking into the design of suitable meta-data structures, including classification and re-classification of objects as well as GoI-splitting policies.

Acknowledgements

This work has been performed partially in the framework of the EU IST project MMAPPS "Market Management of Peer-to-Peer Services" (IST-2001-34201), where the ETH Zürich has been funded by the Swiss Bundesministerium für Bildung und Wissenschaft BBW, Bern under Grant No. 00.0275.

References

[1] H. Balakrishnan, M. Kaashoek, D. Karger, R. Morris, I. Stoica: *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications;* ACM SIGCOMM, San Diego, CA, U.S.A., August 2001.

[2] M. Balazinska, H. Balakrishnan, D. Karger: *INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery;* Pervasive 2002 - International Conference on Pervasive Computing, Zurich, Switzerland, August 2002.

[3] E. Cohen, A. Fiat, H. Kaplan: *A case for associative Peer to Peer Overlays;* HotNets-I, Princeton University, Princeton, NJ, U.S.A., October 28/29, 2002.

[4] A. Crespo, H. Garcia-Molina: *Routing Indices For Peer-to-Peer Systems;* International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, July 2002.

[5] Druschel, Rowstron: *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems;* IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, 2001.

[6] R. Korfhage: *Information Storage and Retrieval;* J. Wiley, New York, U.S.A., 1997.

[7] Q. Lv, P. Cao, E. Cohen, K. Li, S. Shenker: *Search and replication in Unstructured Peer-to-Peer Networks;* 16th ACM International Conference on Supercomputing (ICS'02), New York, U.S.A., June 2002.

[8] P. Maymounkov, D. Mazieres: *Kademlia: A Peer-to-peer Information System Based on the XOR Metric*; 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, MA, U.S.A., March 2002.

[9] J. Mischke, B. Stiller: *Peer-to-peer Overlay Network Management Through AGILE;* IEEE International Symposium on Integrated Network Management (IM), Colorado Springs, CO, U.S.A., March 2003.

[10] J. Mischke, B. Stiller: *Design Space for Distributed Search $(DS)^2$ - A System Designers' Guide*; ETH Zurich, Switzerland, TIK Report Nr. 151, September 2002.

[11] F. Pachet, D. Cazaly: *A Classification of Musical Genre;* Proceedings of Content-Based Multimedia Information Access (RIAO) Conference, Paris, France, 2000.

[12] V. Papadimos, D. Maier , K. Tufte: *Distributed Query Processing and Catalogs for Peer-to-Peer Systems*; Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, U.S.A., January 2003.

[13] M. Prinkey: *An Efficient Scheme for Query Processing on Peer-to-Peer Networks* ; http://aeolusres.homestead.com/files/index.html (August 23, 2002).

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker: *A Scalable Content-Addressable Network;* ACM SIGCOMM, San Diego, CA, U.S.A., August 2001.

[15] S. Rhea, J. Kubiatowicz: *Probabilistic Location and Routing;* 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), New York, U.S.A., June 2002.

[16] M. Ripeanu, A. Iamnitchi, I. Foster: *Mapping the Gnutella Network;* IEEE Internet Computing, Vol. 6, Nr. 1, Jan./Feb. 2002.

[17] C. Rohrs: *Query Routing for the Gnutella Network, Version 1.0;* http://www.limewire.com/developer/query_routing/keyword%20routing.htm (September 3, 2002), May 16, 2002.

[18] B. Silaghi, S. Bhattacharjee, P. Keleher: *Routing in the TerraDir Directory Service*; SPIE ITCOM'02, Boston, MA, U.S.A., July 2002.

[19] K. Sripanidkulchai, B. Maggs, H. Zhang: *Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems;* INFOCOM 2003, San Francisco, U.S.A., April 2003.

[20] K. Sripanidkulchai: *The popularity of Gnutella queries and its implications on scalability;* http://www-2.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html (available on Feb. 03, 2003).

[21] B. Yang, H. Garcia-Molina: *Improving Search in Peer-to-Peer Networks;* Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, July 2002.

[22] B. Zhao, J. Kubiatowicz, A. Joseph: *Tapestry: An infrastructure for fault-tolerant wide-area location and routing;* Technical Report UCB/CSB-01-1141, Computer Science Division, U.C. Berkeley, U.S.A., April 2001.