Efficient Protocol Specification and Implementation for a Highly Scalable Peer-to-Peer Search Infrastructure

J. Mischke[1], B. Stiller[1,2]
[1] TIK, Swiss Federal Institute of Technology, ETH Zurich, Switzerland
[2] IIS, University of Federal Armed Forces Munich, Germany
{mischke,stiller}@tik.ee.ethz.ch

Abstract

While scalable mechanisms for lookup of unique IDs in peer-to-peer (P2P) systems have been found, scalability remains an issue for P2P keyword search. Therefore, a new solution, the SHARK algorithm, has been proposed. Constructing a symmetric redundant hierarchy of nodes and information objects allows for efficient query routing toward small semantic clusters of peers. To show this algorithm's applicability, a detailed specification of the SHARK protocol and a thorough evaluation of its performance is provided. In addition to proving the validity and technical feasibility of the algorithm, this forms the basis for large scale use in several P2P applications. While providing rich keyword search functionality, it is shown that SHARK can easily achieve four orders of magnitude scalability improvement over Gnutella-like networks, greatly outperforming approaches like expanding ring search or associative overlays.

Keywords

Peer-to-Peer, Search, Service Discovery, Scalability, Overlay Network Protocol

1. Introduction

P2P networks, built from nodes at the edge of a network without central servers, strongly depend on a decentralized object search infrastructure. In fact, highly popular filesharing applications like KaZaA, eDonkey, Gnutella, at their core, are content file search complemented with technically rather simple one-to-one or more advanced parallel download functionality. Many other applications of P2P search exist as well. In the global grid, participants have to search vast distributed content repositories for the data they need, often raw measurement data. For data analysis, appropriate computing resources have to be found that match a user's needs in terms of speed, quality of service, or available capacity. In P2P trading applications, peers post or advertise on the one hand, and search on the other hand, classified ads. Instant messaging requires search for a user, *i.e.*, his location and presence information, knowing an alias. In P2P collaboration systems, peers mutually search for documents or data. PGP (Pretty Good Privacy) necessitates a user's public encryption key to be found via his email address. Finally, many expert or knowledge market

places exist already, where users can search for advice on a certain problem they are facing. While technically usually implemented as client/server systems, the dual role of users as experts and advice seekers and their direct interaction make a P2P solution seem very well-suited.

Solutions to P2P search have been developed, but two major issues remain. On the one hand, users usually demand rich search capabilities, *i.e.*, search based on object meta-data in potentially several dimensions, range searches, or keyword search with wildcards. Many systems, however, only perform object lookup based on an exact identifier. On the other hand, some systems like most current filesharing applications allow search by simple keyword strings and it would be technically feasible to enhance them with meta-data and range search capabilities. However, scalability remains a serious issue, as communication between and collaboration of large numbers of peers is required.

SHARK [9] (Symmetric redundant Hierarchy Adaption for Routing of Keywords) is a highly scalable P2P search infrastructure. It provides rich search capabilities including keyword, meta-data, and range searches. At the same time, it achieves scalability by establishing a multi-dimensional symmetric redundant hierarchy of meta-data categories and arranging nodes and objects into it. SHARK can be used in most P2P applications, particularly trading and filesharing.

This paper presents related work in Section 2 and briefly summarizes the SHARK algorithm from [9] in Section 3. It then contributes a detailed specification of the protocol in Section 4. Section 5 evaluates the protocol implementation and final conclusions are drawn in Section 6.

2. Related Work

There are essentially two fundamentally different approaches to P2P search: those based on random topologies and table structures and structured approaches or distributed hash tables (DHTs). Random approaches like Gnutella, expanding ring search, random walk [6], potentially enhanced by interest-based shortcuts [15], semantic gossiping or Bloom filter-based object location storage [12], [3] are inherently not scalable: they conceptually rely on contacting all or a major part of all nodes in the network in order to retrieve information. Structured DHT approaches, like Pastry [4], Tapestry [17], AGILE [7], CAN [11], or Chord [1] are highly scalable. However, they rely on an ordering of objects based on numerical IDs and are hence limited to lookup of these exact IDs; keyword or even rich meta-data search remains impossible.

For scalability reasons, SHARK also builds a structured topology, but departs from the notion of numerical IDs. It constructs a multidimensional 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.

Perhaps most closely related is TerraDir [14], which builds a keyword tree. However, it appears impossible to define the complete ontology of object descriptions down to the leaves that TerraDir assumes, and its hierarchical nature and, hence, different roles of nodes, makes it unsuitable for pure P2P applications. Due to space

constraints, we refer the reader to [9] and [8] for a more detailed discussion of related work and a comprehensive design space and survey of P2P search.

3. The SHARK Algorithm

SHARK applies a hierarchical structure of meta-data categories. Nodes are assigned to categories in Groups-of-Interest (GoI) and, above the group level, connected in a symmetric redundant hierarchy. Object links are advertised to corresponding meta-data categories/GoIs such as to subsequently allow efficient query routing along the hierarchy.

Terminology

Consider a peer-to-peer network consisting of a set of *nodes* $SN = \{N_i | i = 1..n\}$ connected via a set of *links* $L = \{\ell_{ij}\}$. We call $SN_i = \{N_j | \exists\ \ell_{ij}\}$ the *neighborhood* of node N_i. Each node N_i stores a set of *objects* $O_i = \{o_{ik}\}$. The peer offering an object is called the object *owner*, a peer providing a link to an object to alleviate search is called *indexer*.

Meta-data

An object is described through *meta-data* $M_j^{\ell d} = M(O_j)$, where we assume meta-data to be hierarchical with several levels ℓ and dimensions d. It is important, however, that an arbitrary string expression M_j^0 can be used to describe the object without any constraints in the lowest-level meta-data category. Figure 1 (a) illustrates this meta-data structure for a P2P trading example. M^{11} yields the top-level object categorization, that is further divided into subcategories M^{21}. In the second dimension M^{12}, objects are classified by price range, then (M^{22}) by year of construction. M^0 is the search string, *e.g.*, '320d, Leather'. Other applications can simply develop other schemes for different scenarios like categorizations of objects in medicine or jurisdiction or music genre in filesharing.

Overlay Topology

SHARK arranges nodes into a multidimensional symmetric redundant hierarchy (SRH). The overlay topology exactly matches the structure of the object or query meta-data such as to exploit the alignment for query routing. Figure 1 (b) illustrates the topology as well as the request routing procedure. Each node is assigned to a *group-of-interest* (GoI) according to the objects it stores and to its prior request behavior. Each GoI represents a leaf in the hierarchy or a meta-data category.

Above the GoI level, each node knows along each dimension at least one representative from each other GoI. This continues up to the root, where each node knows in each dimension one representative from all meta-data categories. In the example, a node in a Cars/10-20K|BMW/98-00 GoI would know one peer each for all other categories like Electronics, Real Estate, etc., one node each for all other

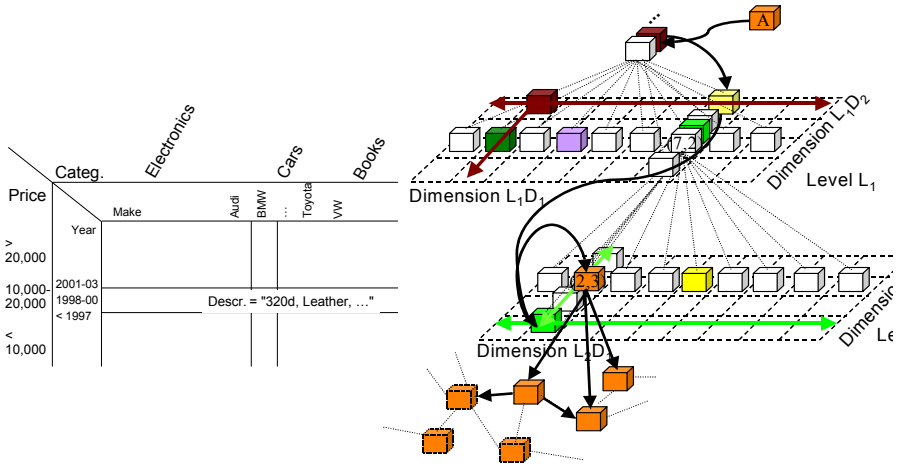

Figure 1. (a) Multidimensional Metadata Hierarchy and (b) Corresponding Overlay

price ranges, as well as within Cars/10-20K one node each for all other sub-categories, *i.e.*, manufacturers, and one node each for 2001-03 and <1997. It is important to note that there are no dedicated servers responsible for higher-level categories or the root position, but that every peer has sufficient routing information and connectivity to assume the role of the root or participate in the routing on any other level. This way, as required for P2P, a symmetric view on the network is achieved while at the same time adding redundancy for increased fault tolerance.

Within the leaf GoIs, peers maintain links to further neighbors SN^R, as indicated in the figure. The overlay network at this stage is, however, unstructured or random.

Search and Query Routing

Search for objects in SHARK is based on query routing. A *query* is defined through a meta-data description M_q of the desired object(s). With the query meta-data structure and the overlay topology aligned and defined as above, queries can efficiently be routed towards relevant content or, more generally, objects. When a node N_A initiates or receives a query, it sends or forwards it, respectively, to all neighbors whose position meta-data exhibits sufficient similarity with M_q on the first level (or, for forwarding, the currently relevant level). 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 some hops can be avoided. The process is repeated until the query reaches the corresponding leaf position in the hierarchy. From there on, it is flooded throughout the GoI along the random power-law network. Every node that caches a link to an object with good similarity to the keyword description part M_q^0 of the query returns that link to the requestor.

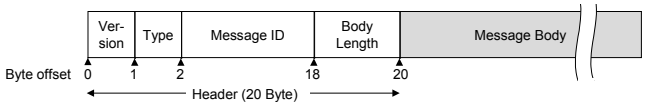

Figure 2. Message Format

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, he can simply use a disjunctive combination to specify the respective $M_q^{\ell d}$. Similarly, he can use wildcards like '*' to initiate an incomplete request and search, *e.g.*, for BMW 320 regardless of the built year. 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 category.

4. Specification of SHARK

Building on the concept described so far, this section summarizes the specification of SHARK. Due to space constraints, the reader is referred to [10] for details.

4.1 Message Format

The message format used in SHARK is shown in Figure 2. The header includes a version field, a type field to distinguish different messages, and a body length indicator as the length of the message body and its fields depend on the message type and actual content being sent. Finally, the header additionally includes a message ID that is 16 Byte long like in Gnutella [2] and can be generated locally on a peer as a random number. The purpose of the ID is to uniquely identify each message for two reasons. First, flooding is used in the random network part of SHARK, requiring the ID for loop detection and for discarding duplicate messages. Second, as receive and send queues will be used for communication, the ID serves to associate incoming replies with previous requests sent.

4.2 Main Functionality

We have identified 9 use cases as most crucial for SHARK and specified them:

- *Object Insertion*: Before query routing can succeed, objects or object links have to be inserted into the network and the correct GoI.
- *Object Removal*: Even though stale links are automatically reported and removed over time in the object transfer use case, some users may want to explicitly revoke object links from the network.
- *Node Insertion*: Nodes have to be carefully inserted into SHARK to successively build up and maintain the multidimensional hierarchy.
- *Node Removal*: When nodes leave the system, a removal process makes sure that the links they are responsible for are transferred to other peers and that their

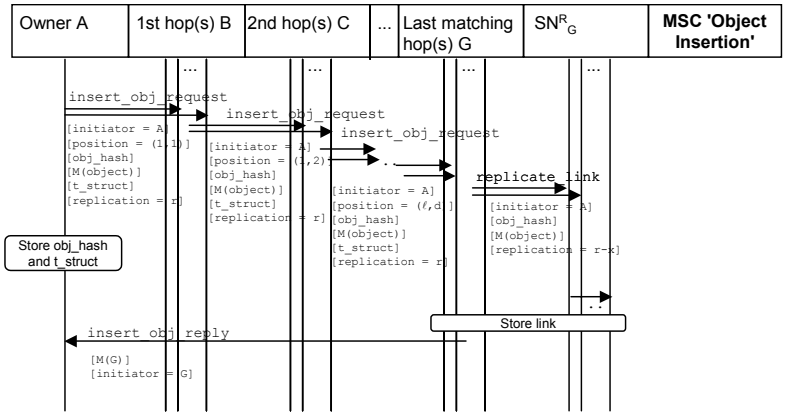

Figure 3. MSC Object Insertion

neighbors' routing tables are updated such as to avoid future routing to a node that is no longer part of the network.

- *Query*: Nodes want to query for objects and receive answers as described in the previous section.
- *Routing Table Repair*: Links may break or nodes die quiet without initiating the node removal process. Routing tables that direct to such links or nodes can be repaired exploiting the redundancy in the system.
- *GoI Split*: As the network grows, GoIs become larger and larger. While one level of hierarchy may suffice for small networks, additional levels are added by splitting existing GoIs into further categories.
- *GoI Insertion*: While the meta-data structure is usually defined by the application up front, it may need to evolve over time. For this purpose, further GoIs, *i.e.*, meta-data categories can be created.
- *Object Transfer*: Once an object matching a query has been found, the requestor will want to have it transferred.

All of these use cases have been specified in [10]; due to space limitations, only object insertion is explained in closer detail here.

Object Insertion

Figure 3 shows the message sequence chart (MSC) for object insertion. At the top of the figure, the parties involved in the use case are shown: object owner and first to last hop in the overlay as well as the last hop's random neighbors. The arrows denote messages to be sent, with the message type on top of the arrow and its fields below. In boxes are process instructions to be run locally on a peer node. The vertical arrangement from top to bottom relates to timing: events in the upper part of the figure occur prior to events below.

When an owner A wants to make an object available, he assigns a SHARK-conform meta-data description M(object) and initiates an `insert_obj_request`. The insert request is routed through the hierarchy analogous to query routing. When it reaches its last hop G in the correct leaf GoI, G stores a link to the object. Further-

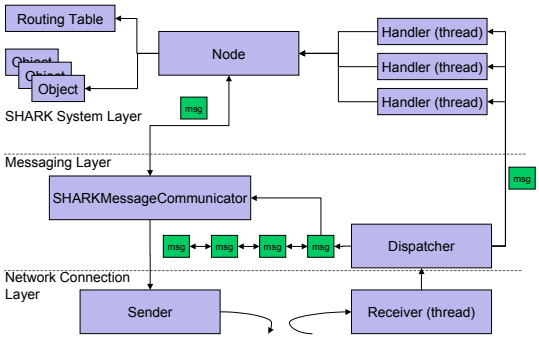

Figure 4. Implementation Module Structure

more, it sends a `replicate_link` message to r neighbors in its random neighborhood SN_G^R to ensure appropriate replication, where r is a replication parameter set by A. G sets a new $r' = 0$ to avoid recursiveness. If $r > |SN_G^R|$, it forwards to the last neighbor a modified request `replicate_link`$(r' = r \angle |SN_G^R|)$. It may happen that the `insert_obj_request` has not yet reached the correct leaf GoI at G, *i.e.*, G has children below the current routing position but none of them matches the request because the correct GoI is still empty. In this case, G simply proceeds as above and inserts the object into its own leaf GoI. The node insertion process described in [10] makes sure that the object link will be moved once a first member of the correct GoI arrives. G confirms object insertion with an `insert_obj_reply` and indicates the GoI that the object has been inserted into by providing its own meta-data.

For the routing, A specifies a threshold t_{struct} for minimum keyword match in the hierarchy. This way, or by leaving some fields in the meta-data description empty, an object link may be routed to and inserted into several GoIs.

5. Implementation and Evaluation

SHARK, as specified above, has been implemented on Windows XP in Java such as to be easily portable to other platforms. The implementation proves feasibility and efficiency of the overall architecture, the communication infrastructure, message handling and dispatching, as well as node insertion, object insertion, and query. A detailed performance evaluation through statistical protocol analysis is given in Sections 5.1 to 5.4.

Figure 4 defines the implementation module structure. To allow for an easy exchange of communication mechanisms and a potential later portability to JXTA, a layered approach has been taken. SHARK functionality is located in the top layer, all SHARK message communication is handled by the second layer which, in turn, builds on the lowest level for communication on the underlying network. Communication is done via UDP datagrams with acknowledgement messages upon reception to detect packet loss.

A 'SHARK Node' passes all messages to the 'SHARK Message Communicator'. The 'Communicator' parses the messages and has them sent as a byte array in a UDP packet via a 'Sender'. A 'Receiver' thread listens for incoming messages and passes them on to a 'Dispatcher'. Acknowledgements for previously sent messages and reply messages are fed into a message queue that is read by the 'Communicator'. Reply messages are forwarded to the 'SHARK Node' for further handling if appropriate. Acknowledgements, in contrast, can directly be handled by the 'Communicator'; the absence of an acknowledgement leads to a re-send of the respective message or to an exception if two consecutive trials fail. The 'Dispatcher' takes care of all incoming requests from other nodes and starts appropriate handler threads. These handlers apply 'SHARK Node' functionality like routing, object link storage, or object transfer to process requests. Each 'SHARK Node' maintains state information, most notably the routing table and the object links it is responsible for as required to appropriately handle requests.

5.1 Multidimensional Hierarchy Structure

The number of nodes, the meta-data categories, and the distribution of nodes to GoIs determine the hierarchy structure created in SHARK. For the disposition of nodes to GoIs, we have modeled two distributions: a uniform distribution (most simple case) and a bi-modal Zipf distribution. It has been observed (cf., *e.g.*, [5]) that document popularity follows a Zipf distribution, and it is reasonable to make the same assumption for category or GoI popularity. 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 [16] 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 [5] and is the average reported in [16], and we set inflection at t_{inf}=1% of possible GoI ranks.

Figure 5 (a) evaluates the effect of node distribution to GoIs on the SHARK hierarchy. The dashed line shows the average number of hierarchy levels assuming a uniform distribution of nodes to GoIs, two dimensions on each level with 8 meta-data categories per dimension (*i.e.*, 64 meta-data categories per level), and GoI splitting at group sizes larger than 500 nodes. Roughly 30,000 nodes can be supported with one hierarchy level, more than a million with two levels. The solid line represents the result for a bi-modal Zipf distribution of nodes to GoIs. As some groups are significantly more popular than other groups, splitting occurs earlier in some parts of the hierarchy, leading to imbalances, where we assumed splitting to result in, again, 8 by 8 subgroups, and a Zipf distribution of nodes to subgroups within a higher-level group. While the error bars in the figure show the maximum and minimum number of levels, the solid line is an average across all GoIs, weighted by their respective sizes in terms of number of nodes. Clearly, a meta-data hierarchy that achieves as uniform as possible a distribution of nodes to GoIs is advantageous. However, even in the more typical Zipf-case, the average level number remains below three even at a million nodes and increases only logarithmically.

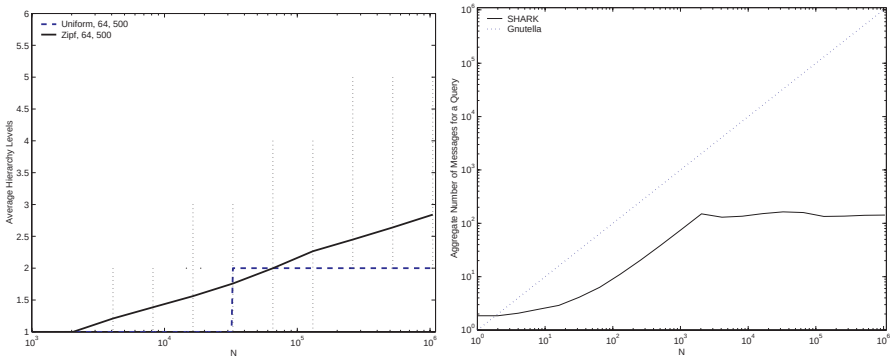

Figure 5. (a) Hierarchy Levels and (b) Aggregate Messages for a Query

5.2 Cost of Queries

We consider the aggregate number of messages necessary to resolve a query as it determines bandwidth needs, the request pathlength as the key driver of latency, and the node degree due to its relevance for processing power and memory demand.

Aggregate Number of Query Messages

The number of messages for a query is $m_{agg} = h_h + n(GoI)$, where 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]. Figure 5 (b) compares m_{agg} for SHARK (solid line) and Gnutella (dotted line), assuming, as usual, Zipf popularity, 64 categories per level, and a maximum GoI size of 500. In addition, it is assumed that the meta-data hierarchy is sufficiently deep to allow for GoI splitting, that the Gnutella reference case has a sufficiently large time to live, and that pruning occurs at a probability p_p=5%. In small networks, SHARK exhibits a linear increase while GoIs on the first level are being filled with nodes that need to be contacted by flooding, achieving roughly an order of magnitude improvement over Gnutella. Once considerable GoI splitting occurs due to further network growth, the bandwidth demand in SHARK grows sub-logarithmic or remains almost constant rather than linear as in Gnutella: while flooding is limited to the maximum GoI size, only 2 additional messages are needed to route through each additional level in the hierarchy. This represents the key achievement of SHARK.

Request Pathlength

The request pathlength PL_R within the entire SHARK is the number of hops in the hierarchy h_h to reach a certain GoI plus the average number of hops $\bar{h}_{GoI}$ within that GoI, averaged across all GoIs. The average obviously needs to be weighted by the

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.

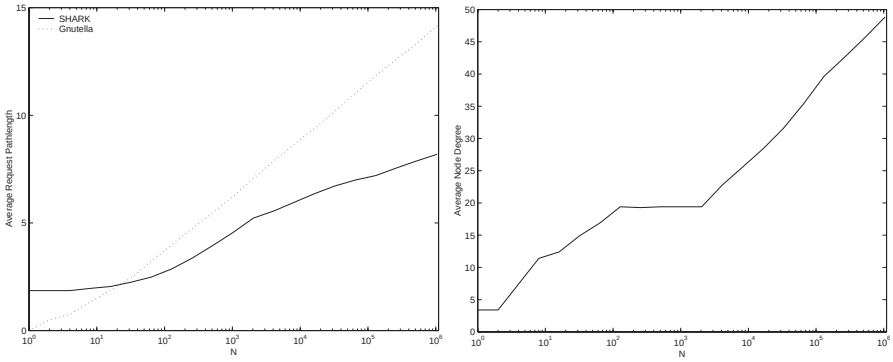

Figure 6. (a) Request Pathlength and (b) Node Degree

relative request frequencies to GoIs so that the larger, more popular GoIs with unfortunately more hops are queried more frequently.

Figure 6 (a) compares the request pathlengths of SHARK (solid line) and Gnutella (dotted line), with the same assumptions as for m_{agg} and a random network node connectivity $\bar{\ell} = 3.4$ [13]. It is shown that SHARK may even outperform the already benign logarithmic behavior of Gnutella. The request pathlength of both, Gnutella and SHARK, will be even lower in power-law networks.

Node Degree

Figure 6 (b) depicts the average node degree for a SHARK node and is also representative of its routing table size. The node degree accounts for 8 nodes on each level and in both dimension plus $\bar{\ell} = 3.4$ neighbors in the random network part. It first grows while the first level of the hierarchy is being filled with nodes to connect to, then remains constant at roughly 20 when the first level is complete, until it finally develops logarithmically as the network grows further and GoI splitting occurs. The node degree remains below 50 at a million nodes.

5.3 Cost of Overlay Network Management

Other than random networks, SHARK incurs additional cost for managing the hierarchical overlay network. We evaluate this cost along the use cases for normal operation: node insertion, node removal, object insertion, object removal, and GoI splitting, and we also take a look at the state information to be kept on each node. For all evaluations, we assume a bi-modal Zipf popularity of GoIs, s=64 categories per hierarchy level, a maximum GoI size before splitting of 500 nodes, a node connectivity $\bar{\ell} = 3.4$, a link redundancy upon object insertion r=3, and an average number of |o|=100 objects per node.

Cost of Node Insertion and Removal

The cost, *i.e.*, the aggregate number of messages to insert a node, is depicted in Figure 7 (a). For reasonably large networks, the cost develops logarithmically up to some 35 messages in systems of one million nodes. A significantly higher cost is

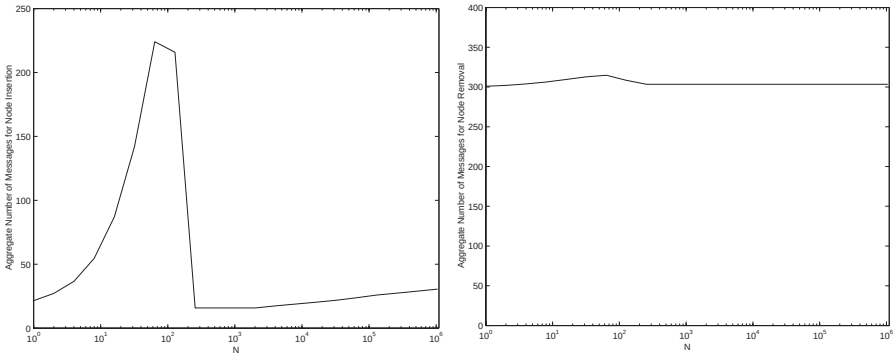

Figure 7. Cost of (a) Node Insertion and (b) Node Removal

incurred in young networks as they grow toward the first GoI splitting. When a new node is the first member of a previously empty GoI, it has to announce its existence throughout the entire parent GoI. Furthermore, some object links may need to be transferred to the now occupied GoI.

In non-growing networks, node removals occur as frequently as node insertions. The cost for node removals includes one transfer message for each object link and one message to inform each neighbor in the random network, largely independent of network size as shown in Figure 7 (b). In small networks, a leaving node may result in the affected GoI becoming empty. In this case, the leaving node has to inform all nodes in the parent GoI of its departure. Also, transferring links becomes slightly more expensive, as it involves link redundancy to be re-established; hence the slightly higher cost up to roughly 100 nodes.

Cost of Object Insertion and Removal

Object insertion results in a number of messages equal to the number of hops in the SHARK hierarchy, h_h, plus one answer message, and further messages to replicate the link for redundancy. Figure 8 (a) .depicts the total assuming a pruning probability p_p=5% as before. It remains constant as long as the hierarchy is only one level deep, and grows logarithmically with the size of the network once GoI splitting occurs. It is very low compared to the cost of queries.

Object removal messages take exactly the same path as queries and, hence, incur exactly the same cost. However, usually, object links are not explicitly revoked at all, but stale links are identified and removed when attempting to transfer objects.

Cost of GoI Splitting

GoI splitting occurs during phases of network growth to add further hierarchy levels. Each event of splitting requires a `split_GoI` message to be flooded to each node in the group. For each new subgroup, one node will become its first member and announce its presence to the other nodes. In order to reach its new target subgroup, each node follows the hierarchy one level further down. Finally, nodes have to transfer part of their links. The number of messages for a GoI split is depicted in Figure 8 (b) as a function of the maximum GoI size. It also shows that a higher number of subgroups (s) leads to increased traffic due to node announcements.

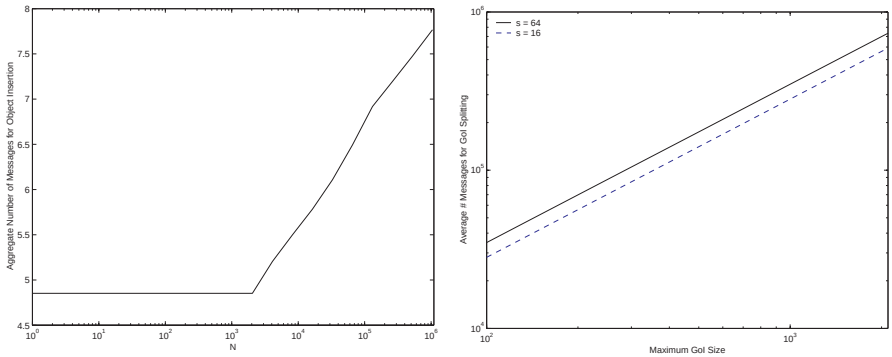

Figure 8. Cost of (a) Object Insertion and (b) GoI Splitting

State Information

The state information to be kept in memory on each node comprises three parts: the additional redundancy in the routing table that does not form part of the node degree, the object links to be stored, and the meta-data hierarchy. The redundancy in the routing table is (at most) as large as the node degree. The link information contains for each link, with three redundant links per object, a 16-byte hash code, the IP address and port of the respective owner, a categorization into the meta-data structure, and a keyword description M^0. The meta-data hierarchy requires a description to be stored for each GoI and each higher-level category. The total state information per node is plotted in Figure 9 (a), conservatively assuming a meta-data structure that is everywhere as deep as the maximum number of levels in the hierarchy and 20 Bytes per category description. It remains well below 1 MByte for networks as large as 1 million nodes.

5.4 Total SHARK Bandwidth

Having evaluated the cost of query activities and overlay network management separately in the previous two sections, it is now time to bring both together and assess the overall bandwidth demand of SHARK. Figure 9 (b) compares the aggregate number of messages per node and day or second of Gnutella and SHARK in three

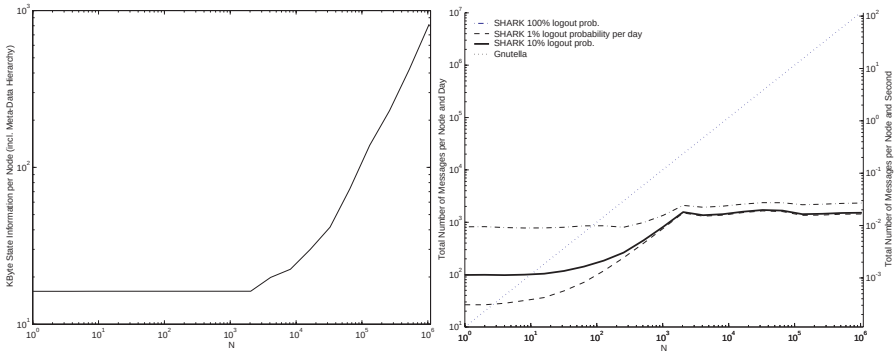

Figure 9. (a) State Information and (b) Total SHARK Bandwidth

scenarios. Each scenario assumes a query frequency of 10 queries per day and node, while the rate of node departures and re-insertions varies. In the first scenario (dash-dotted line), it is set to one per day, in the second one (solid line) 0.1 per day, and finally 0.01 per day in the third one (dashed line). It is easy to recognize the additional network management overhead incurred as the rate of node joins and departures increases. However, even at one logout per node and day, the management overhead remains relatively small compared to the query activity. SHARK clearly exhibits an improvement of several orders of magnitude over Gnutella. It also demonstrates logarithmic or even sub-logarithmic scalability as long as the meta-data hierarchy can be defined and applied in sufficient depth.

6. Summary and Conclusion

The proposed system SHARK (Symmetric redundant Hierarchy Adaption for Routing of Keywords) defines a novel approach to P2P keyword search. By restricting query flooding to small semantic clusters or GoIs, vast performance improvements between one and four orders of magnitude compared to random networks like Gnutella are achieved in networks ranging from a thousand to a million nodes. With the assumptions from the previous section, both object insertion and node insertion scale logarithmically and usually require less than ten or a few tens of messages to be sent, keeping overlay network management cost low. Queries can efficiently be resolved so that both, request pathlength and query messages develop at most logarithmically with network size. Query bandwidth has even been shown to remain roughly constant from a few thousand to a million nodes in the network.

While this performance trend would persist up to billions of nodes and beyond, usability becomes an issue for too large network sizes. Whereas users are expected, depending on the application, to easily navigate through the two to five levels of meta-data hierarchy at a million nodes, an additional two to four levels required to support hundreds of millions of peers may turn out to be too complicated.

SHARK is expected to be used in a variety of applications. First and foremost, large-scale filesharing systems command large user bases and would benefit from SHARK's scalability improvements. Furthermore, SHARK is well suited for any application where the establishment of semantic clusters and meta-data categories is viable. This includes, *e.g.*, P2P trading systems, where the categorization is identical to ad classification in 'classified ads', or expert and knowledge marketplaces with their ontology of topics.

Going forward, it is intended to build a P2P trading application on top of SHARK. In addition, a detailed investigation of security issues relating to structured P2P lookup and search approaches is not yet available and would be helpful to assess, in particular, the threat of denial of service attacks.

Acknowledgments

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., Aug. 2001.

[2]. Clip2: *The Gnutella Protocol Specification v0.4*; http://www.clip2.com/GnutellaProtocol04.pdf (in May 2002).

[3]. F. Cuenca-Acuna, C. Peery, R. Martin, T. Nguyen: *PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities*; HPDC-12, Seattle, WA, U.S.A., Jun. 2003.

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

[5]. R. Korfhage: *Information Storage and Retrieval;* J. Wiley, NY, U.S.A., 1997.

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

[7]. J. Mischke, B. Stiller: *Peer-to-peer Overlay Network Management Through AGILE;* Int'l Symp. on Integrated Network Management (IM), Colorado Springs, CO, U.S.A., Mar. 2003.

[8]. J. Mischke, B. Stiller: *A Methodology for the Design of Distributed Search in P2P Middleware*; IEEE Networks, Vol. 18, No. 1, Jan./Feb. 2004.

[9]. J. Mischke, B. Stiller: *Rich and Scalable Peer-to-Peer Search with SHARK*; Advanced Middleware Services (AMS), Seattle, WA, U.S.A., Jun. 2003

[10].J. Mischke, B. Stiller: *Specification of a Scalable Peer-to-Peer Search Infrastructure*; ETH Zurich, Switzerland, TIK Report Nr. 176, July 2003.

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

[12].S. Rhea, J. Kubiatowicz: *Probabilistic Location and Routing;* INFOCOM, New York, U.S.A., Jun. 2002.

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

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

[15].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., Apr. 2003.

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

[17].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., Apr. 2001.