Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Jan Mischke, Burkhard Stiller

Specification of a Scalable Peer-to-Peer Search Infrastructure

TIK-Report
Nr.176, July 2003

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Jan Mischke, Burkhard Stiller:
Specification of a Scalable Peer-to-Peer Search Infrastructure

Version 1, July 2003
TIK Report Nr. 176

Computer Engineering and Networks Laboratory,
Swiss Federal Institute of Technology (ETH) Zurich

Institut für Technische Informatik und Kommunikationsnetze,
Eidgenössische Technische Hochschule Zürich

Gloriastrasse 35, ETH-Zentrum, CH-8092 Zürich, Switzerland

Specification of a Scalable Peer-to-Peer Search Infrastructur

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

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

E-Mail: [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. One solution, the SHARK algorithm, has been derived and outlined in [16]. In this document, we provide a detailed specification of the SHARK protocol and a thorough evaluation of its performance. While providing rich keyword search functionality, it is shown that SHARK can easily outperform Gnutella-like networks by four orders of magnitude.

Keywords: *P2P, Peer-to-Peer, Search, Scalability, Specification*

1 Introduction

While the concept of peer-to-peer networking dates back to the origins of the Internet with Usenet as the most prominent early P2P application, the world wide web that brought Internet usage to the masses featured predominantly client/server applications. Now, P2P networks are proliferating rapidly again in areas like P2P computing (with Platform Computing, Data Synapse, and United Devices constituting some of the major players), collaboration (Groove, Next Page, Consilient), trading (OpenWebs, WorldStreet), network testing (Porivo, Prompt2U), but maybe on the largest scale in file-sharing (Gnutella, KaZaA, eDonkey, Overnet).

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 shutdown 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.

In contrast to common perception, P2P networks are highly extensible, but scalability is a serious issue. Extensibility describes the ease of extending a network such as to accommodate new users and additional demand. Scalability demands that the efficiency of a network remain high at very large scales. The composition of P2P networks out of edge nodes or peer resources only makes them highly extensible: new resources are added along with every new user or peer. Scalability, however, is difficult to achieve. Completely decentralized search in a network with potentially millions or even billions of peers usually incurs vast communication and connection overhead limiting the efficiency of large networks. Particularly for home or dialup users, bandwidth is still the most expensive and scarcest resource. The bandwidth overhead incurred for the aggregate number of messages that have to be exchanged for a search request thus gains particular importance. Memory and processing power overhead is to a good degree determined by the amount of state information to be kept on each node, particularly for routing information and connections to other nodes in the network - remember that the number of possible connections between peers grows with the factorial of the number of nodes.

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). The IP end-to-end connectivity of peers can be used to establish and maintain virtual links between peers. The entirety of virtual links and peer nodes forms an overlay network on top of the Internet. DHTs construct highly structured overlays like rings, hierarchies, or Euclidean spaces. They subsequently assign (usually numerical) names to both, nodes and objects, and establish an order of names following the overlay structure. Once such an order of nodes, objects, and virtual links is established, the structural information can be exploited to efficiently route lookup requests to the nodes where the objects are stored.

Unfortunately, current DHTs are thus limited to pure lookup of exactly specified unique names. 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.

As an outline to this document, Section 2 gives an overview over related work before Section 3 recaps the SHARK algorithm as presented in [16]. Section 4 identifies the most important use cases and gives a detailed specification. After briefly presenting the implementation architecture in Section 5, a thorough evaluation is provided in Section 6 before Section 7 concludes.

2 Background and Related Work

A complete design space and methodology to design distributed search systems has been presented in [15] and led to the development of SHARK as one of the few possible entirely novel approaches to P2P search. It is important to separate functional and structural aspects of P2P lookup and search systems. In the functional dimension, there are three essentially different 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 [4], mapping keywords onto hash IDs that may or may not exist in the network. Second, there are lookup or name routing mechanisms that take unique names as an input and lookup or route towards the corresponding objects; examples include distributed hash tables like Chord [2]. Finally, there are keyword lookup/routing schemes that combine the first two systems and find or 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.

As SHARK may show structural or functional similarity with one or the other existing system, but not both at the same time, it is an entirely novel concept filling a major and interesting gap in the design space.

2.1 Functionally Related Systems

There are several other systems applying SHARK's combined keyword lookup/routing approach. Napster applies a central server and is thus not well suited for pure P2P networks.

Random network approaches like Gnutella, expanding ring search [30], or random walk [12] 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 [27] 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 [7] 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 [30]. 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. Prinkey [19] proposes Bloom filters in tree topologies with aggregated signatures of a branch, *i.e.* the Bloom filter bits represent the hashed keywords present in a tree branch. LimeWire modifies the proposal to cope with arbitrary topologies by adding the number of hops to a resource when propagating the keyword routing information [24]. Crespo and Garcia-Molina [8] suggest to store and propagate the number of matching documents for each keyword, either together with the number of hops to a document (hop count routing indices), or weighting the number of documents with a cost function depending on the distance (exponentially weighted routing indices). 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 [26], 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 [18], 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 unsuitable for pure P2P applications.

2.2 Structurally Related Systems

AGILE [14] 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 [10], Tapestry [31], and Kademlia [13] essentially build one-dimensional symmetric redundant lookup hierarchies. In Pastry [10] and Tapestry [31], 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 [13], 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. All three of them are entirely built on the notion of unique hash-based numerical IDs for objects and cannot be applied for keyword search.

Chord [2] hashes resource names and node IP addresses to a 128-bit ID. The IDs are arranged in a circle with the predecessor node of a resource ID being responsible for providing the resource or a link to it. Fingers are used as short-cuts to prevent the name routing mechanism from moving around the circle in unit steps. In CAN (Content Addressable Network, [20]), 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 easily 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. Section 3.1 defines important concepts and outlines the scenario we assume. Sections 3.2 and 3.3 define and illustrate the meta-data and the overlay network structure, respectively. Both are well-aligned to subsequently enable efficient search and query routing (Section 3.4). Finally, the actual construction and management of the overlay network is described in Section 3.5.

3.1 Scenario and Definitions

Consider a peer-to-peer network consisting of a set of *nodes* $SN = \{N_i | i = 1..n\}$ connected via a set of *links* $L = \{\ell_{ij}\}$ (cf. Figure 1). We call $SN_i = \{N_j | \exists\, \ell_{ij}\}$ the *neighborhood* of node N_i. Each node N_i stores a set of *objects* $O_i = \{o_{ik}\}$ that constitute to the unique objects in the entire P2P network $O_{tot} = \bigcup_{i \le n} O_i =: \{o_j\}$, where $|O_{tot}| \le \sum_{i=1}^{n} |O_i|$ due to replication of identical objects onto several nodes. An object is described through *meta-data* $M_j = M(O_j)$. The peer offering an object is called the object *owner*, a peer providing a link to an object to alleviate search is called *indexer*.

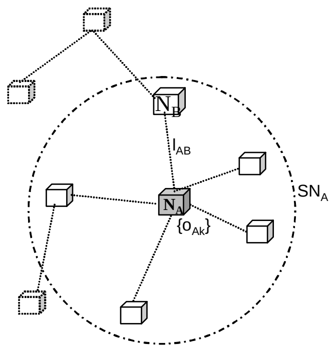


Figure 1: SHARK: Terms and Definitions

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 $M_j = (M_j^1, M_j^2, .., M_j^{\ell}; M_j^0)$. ℓ is the number of levels in the hierarchy, M_j^1 yields a first-level categorisation of the object while $M_j^2, .., M_j^{\ell}$ 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,

$$
\begin{aligned}
M_j^1 &= (M_j^{11}, M_j^{12}, .., M_j^{\ell d_1}) \\
M_j^2 &= (M_j^{21}, M_j^{22}, .., M_j^{2 d_2(M_j^1)}) \\
&\dots
\end{aligned}
$$

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

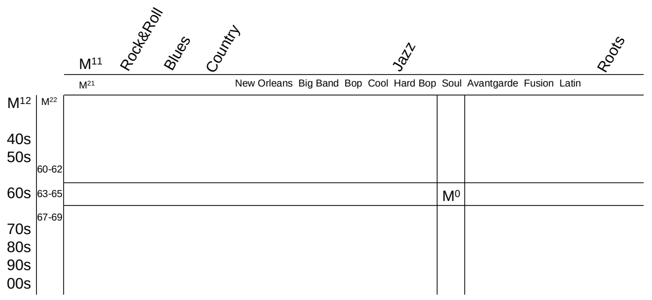


Figure 2: Multidimensional Meta-Data Hierarchy (illustr.)

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.

Other applications can simply develop other 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. Figure 3 illustrates the topology.

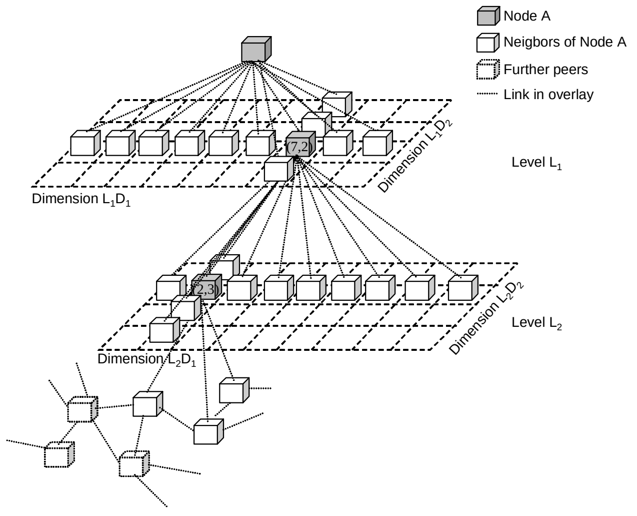


Figure 3: Multidimensional Symm. Redundant Hierarchy

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). Each GoI represents a leaf in the hierarchy.

A Symmetric Redundant Hierarchy (SRH) is constructed as follows. 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 would then be a member of the GoI on leaf position $P_A = (7, 2, 2, 3)$. The SHARK SRH adds redundancy to the hierarchy 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. 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 positions $P_A^{(1)} = (p_A^{11}, p_A^{12})$, 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:

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

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 and ports 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 [23]. It would be possible to further optimise the networks within the GoIs through associative overlay network techniques or the like. This, however, can be done separately from this work.

3.4 Search and Query Routing

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*

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

where $0 \le S(M_a^{jl}, M_b^{j\ell}) \le 100\%$ 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. [11], p.300ff, p.83).

Note that in the SHARK system presented throughout this document, the subset relation in the query answer definition could be replaced through an equality; future extensions, however, may incorporate possibilities to retrieve partial results to further increase scalability in situations with frequent replication.

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 $Q(M_q,t_{struct},t_{rand})$, 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 some hops can be avoided. The 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,$\ell_c$,$\ell_c$) {
        if ($d_c$==d($\ell_c$) && $\ell_c$==$\ell$) {
              flood(Q);
              exit,
        }     // last level and dimension reached
        else if ($d_c$==d($\ell_{c)}$) { // last dimension on level
              $d_c$=0;
              $\ell_c$++; // set dimension 0 on next level
        }
        $d_c$++;
        For N in $SN_A{}^h$($\ell_c$,$d_c$) // Neighbors on ($l_c$,$d_c$)
              if S[$M_q{}^{\ell c,dc}$, M($P^{\ell c,dc}$(N)] >= $t_{struct}$ {
              // Sufficient similarity with position?
                    Send(Q,$\ell_c$,$d_c$) to N;
              }
        }
        if S[$M_q{}^{\ell c,dc}$, M($P^{\ell c,dc}$($N_A$)] >= $t_{struct}$
        // can move down hierarchy on same node
              forward(Q,$\ell_c$,$d_c$); // 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 [12]. 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 [22], [24]. However, in order to ensure maintainability of the system, this approach should only be applied when the rate of object insertion and removal is significantly lower than the query rate. All of these improvements are orthogonal to the SHARK concept and will not be considered in more detail here.

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 $M_q^{j\ell}$, provided the similarity metric has been appropriately chosen to support combinations. Similarly, he 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.

It is obvious that for the query routing to work, object links previously have to be correctly integrated into the network and to be maintained as nodes join or leave the overlay. This will be described in detail in Section 4.5.

3.5 Adaptive Semantic Clustering

The categorisation strongly relates to the specific application. For instance, an extensive music categorisation has been proposed in [17], and ways to automate the classification process are explored in [1]. 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 categorisation 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 $N_{A,vn}$ into an arbitrary child GoI of $M^{j\ell}$, using the same procedure as for real node insertion. Vice versa, whenever the number of identical meta-data $M^{j\ell}$ 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. As in AGILE, the use of virtual nodes serves two purposes. First, it allows SHARK to semantically adapt over time; nodes move towards the content or objects they like and request. Further queries will be served more efficiently as the requestor N_A can initiate a query through its virtual node $N_{A,vn}$ 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 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.

4 Specification of SHARK

Building on the concept described so far, this section gives a detailed specification of SHARK. We first define the routing table structure of SHARK (Section 4.1) and the general message format (Section 4.2) before we analyse use case (Section 4.3 - 4.11): node insertion, node removal, object insertion, object removal, query (incl. query answer), routing table repair, GoI split and GoI insertion, and actual object transfer. Finally, in Section 4.12, we define all messages of the SHARK protocol in detail, including their fields and formats.

4.1 Routing Table

Figure 5 illustrates the structure of the routing table (RT). A variable number of levels can be accommodated. For each

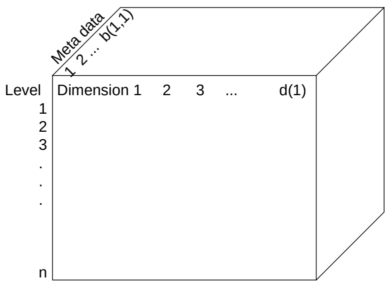


Figure 5: Routing Table

level, there are one or more dimensions, and each level ℓ and dimension d holds up to b(ℓ,d) different meta-data categories. On each level ℓ, the number of meta-data categories that an actual node A's routing table has to deal with depends on A's meta-data description on the previous level ℓ-1; in the example, a node in the "rock&roll" category may have to support more or fewer sub-genres than a node in the "jazz" category.

Each field (ℓ,d,j) in the routing table comprises three data items: the corresponding meta-data description $M_j^{\ell d}$ and two alternative next hops. The choice of two next hops creates additional redundancy in the system for increased fault tolerance. For a node to be stored as next hop in a field (ℓ,d,j), it has to match the current node A's meta-data descriptions in all previous dimensions and on all higher levels.

4.2 Message format

The message format used in SHARK is shown in Figure 6. The header includes a version field, a type field to distin-

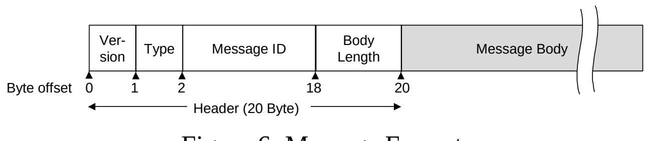


Figure 6: Message Format

guish 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 [6] and can be generated locally on a peer by, *e.g.*, a random number generator or by hashing the message concatenated by the initiator address and a time stamp. 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.

4.3 Node Insertion

Figure 7 shows the message sequence chart (MSC) for node insertion in SHARK. At the top of the figure, the parties involved in the use case are shown: joining node and first to last hop in the overlay as well as the last hop's potential children and its 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. Usually, several different cases have to be distinguished for a use case, *e.g.*, whether an error occurs or not. These cases are named 'legs' in the MSC and are separated by dotted lines.

New nodes can easily join the SHARK overlay using essentially the same routing mechanisms as queries. When a node A wants to join, it sends an `insert_node_request` with the first level and dimension meta-data description of its own GoI to an arbitrary node in the system about which it learned out of band. The contacted node B returns an `insert_node_reply` and transfers its first routing table row, *i.e.*, the row of meta-data categories and next hops on the first level in the first dimension. As A has no means of handling potential faults in the routing table yet, it is up to B to first test the most critical entry. Hence, before sending the reply, B matches A's meta-data with its routing table fields to determine the next hop for node insertion C. It tests if C is alive with a ping message and pong reply. Should C be down, B tries the redundant entry in its RT or, if necessary, proceeds with the RT repair mechanism described in Section 4.8.

In order to avoid frequent insertion and removal of unstable nodes, A connects to the network through B as a proxy for a time-out period τ (see also Section 4.7), before being fully integrated into the overlay network. This procedure helps keep the maintenance overhead low [9]. Measurements suggest that in Gnutella-like networks, roughly 50% of sessions last for less than an hour, whereas the longer sessions tend to last for many hours or days [25]. Once the time-out has been exceeded, B integrates A into its RT, either filling the redundant empty position if still empty or replacing the least-recently seen alive entry. A, in turn, copies the entire RT row it received to set up the first row of its own RT.

The process then continues by A sending an `insert_node_request` to C, providing the first-level, second-dimension meta-data description of its own GoI. A

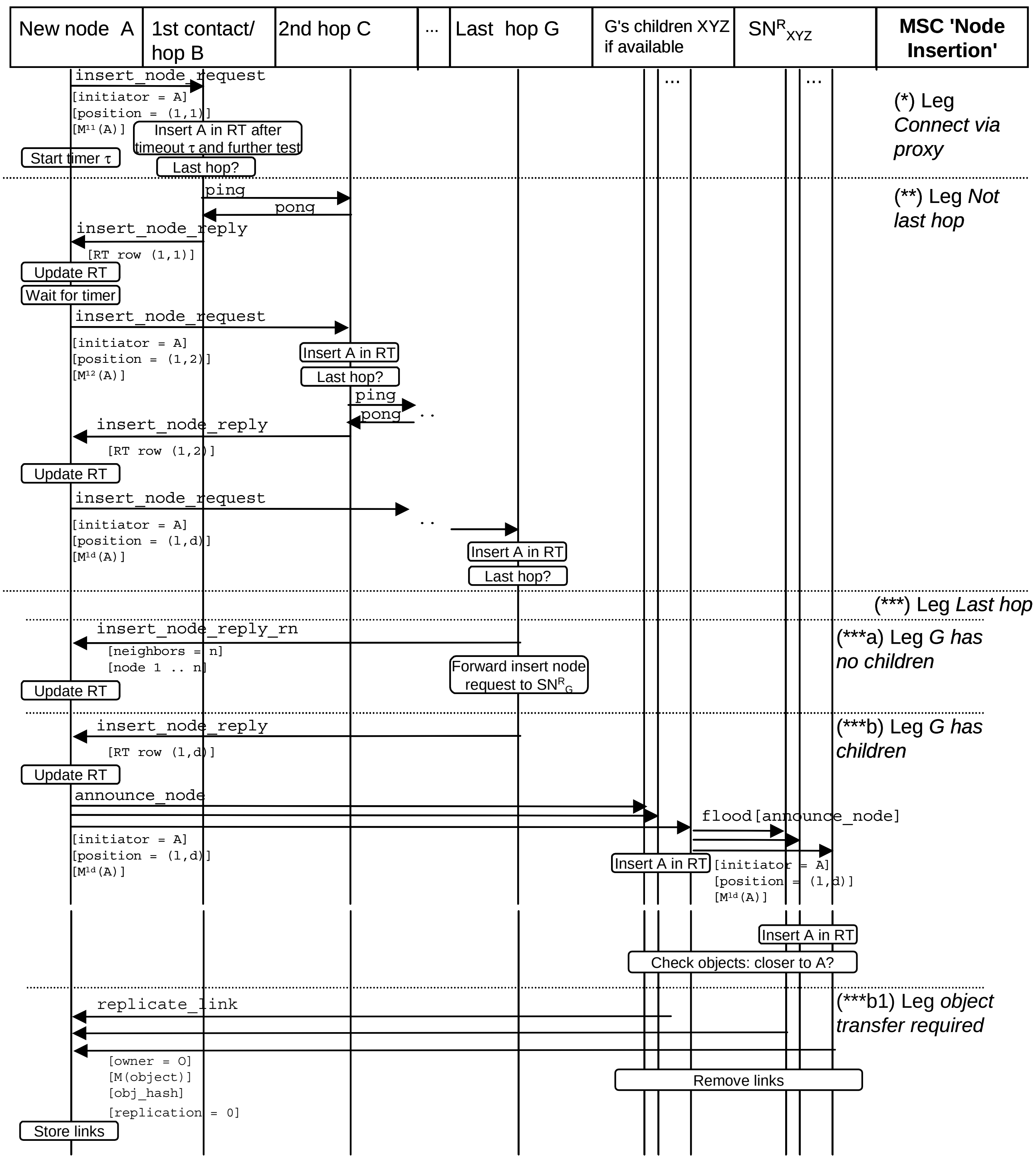


Figure 7: SHARK MSC Node Insertion

also indicates the position(ℓ,d) currently relevant for routing so that C be able to determine which RT row to return and to use for further routing/testing of entries. As before, C integrates A into its RT and A copies C's RT row. The routing for node insertion continues until it reaches the last hop G on a position(ℓ_G,d_G) that has no suitable entry in its RT any more to match A's meta-data description $M_j^{\ell_G d_G}$ on that level.

If position(ℓ_G,d_G) represents G's leaf GoI, *i.e.*, G has no further children in its RT below that position, A is integrated into the random network of that GoI through an `insert_node_reply_rn`, transmitting G's neighbors in the random network SN^R_G to A. Furthermore, G forwards A's `insert_node_request` to all nodes in SN^R_G so that they learn about A and integrate it into their random neighborhood (illustration simplified in Figure 7). A selects random nodes from the various `insert_node_reply_rn` it receives to build up its own random neighborhood SN^R_A.

If G has further children but no match for A, A becomes the first node to fill a previously empty GoI. As usual, G transmits its last RT row so that A can integrate it. A then has to inform its new siblings, *i.e.*, all other children XYZ of G below position(ℓ_G,d_G) through an `announce_node` message so that they learn of A's existence and subsequently be able to correctly route to A's GoI. XYZ flood the `announce_node` message throughout their respective GoIs, and every member updates its RT accordingly. Finally, objects may have previously been inserted that closer match A's GoI than their current indexer's one. The nodes informed of A's arrival hence check their object links and transfer them to A if required. A `replicate_link` message is used for this purpose rather than an `insert_object_request` as no routing is necessary and also to avoid an unnecessary `insert_object_reply` message.

As a possible design alternative, nodes could recursively forward the `insert_node_request` to the next hop rather than sequentially relying on node A. However, this would significantly complicate the issue of inserting A only after a time-out, and it would also require to send A's potentially long meta-data description entirely in each message. Furthermore, A needs a reply from each hop anyway. We will use the recursive message forwarding for query and object insertion, where the concerns described above do not apply in the same way.

During the entire node insertion process, there will usually be considerable choice of possible neighbors. In order to minimize latency, the overlay network should be mapped as closely as possible to the physical network [32]. It is possible to adapt network proximity mechanisms like [5] for use in SHARK.

4.4 Node Removal

It will help stabilise SHARK if a node correctly removes itself from the network rather than dying quietly. Contemplate Figure 8 for a specification. A leaving node A has to transfer the object links it has and inform relevant neighbors of its departure. If A is well integrated into a GoI and has a non-empty set of neighbors in the random network part ($SN_A^R \neq \emptyset$), it simply sends its object links to arbitrary neighbors within SN_A^R with a `replicate_link` message, setting the initiator of the request to the object owner so that the link stored will point to the correct node. Subsequently, A notifies its departure to its neighbors with a `remove_node` message. The neighbors can then choose to establish new links among themselves to compensate A's leave. Should A be the last or only member of its GoI, it first informs all sibling GoIs of its departure so that they can delete A from their RT and so that they know that this GoI has become empty. Then, A sends `replicate_link` messages to its siblings for storage of its links.

4.5 Object Insertion

In order for an object to be found, it is required that it has been inserted or published to that GoI before (Figure 9). 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 or node insertion routing. When it reaches its last hop G in the correct leaf GoI, G stores a link to the object. Furthermore, it sends a `replicate_link` message to r neighbors 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`$(r' = r - |SN_G^R|)$. 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 Section 4.3 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.

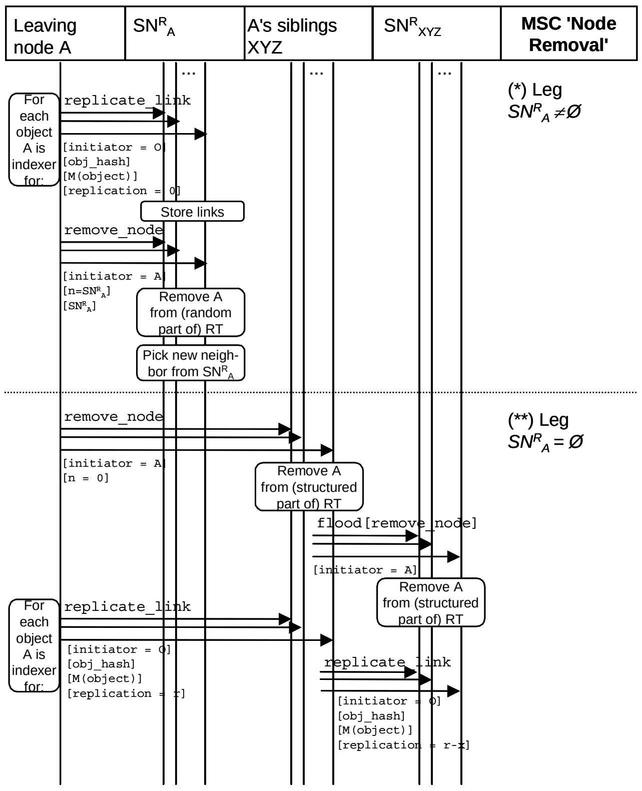


Figure 8: MSC Node Removal

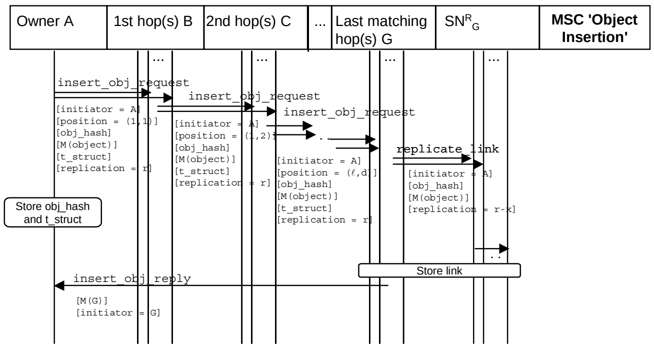


Figure 9: MSC Object Insertion

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. A stores the t_{struct} it set and the object hash that uniquely identifies the object for potential later removal (see next section).

4.6 Object Removal

Reciprocal to the object insertion, an owner A may want to stop offering a certain object. A can choose to just stop the offering; the object transfer process described in Section 4.9 will make sure that all links to the object will be removed over time. Alternatively, A can explicitly revoke the object as specified in Figure 10. A starts routing a `remove_object` message through the network, specifying the same meta-data and t_{struct} as for the insertion to make sure that the same paths are taken. Once the message reaches its last hop G, G floods it throughout its GoI. For the flooding, it is sufficient to identify the object links to be removed by the object hash and the owner. In analogy to the object insertion, should G be the last hop for the request yet still have children further down the hierarchy, G forwards the `remove_object` message to all children that, in turn, flood it throughout their GoIs.

4.7 Query

Query routing has already been explained in Section 3.4. For completeness, Figure 11 provides the corresponding message sequence chart. Requestor A specifies the query through its meta-data description including the keyword string M^0 and starts routing it through the overlay. The query may be duplicated if multiple matches occur on a level. Once a `query` reaches the last matching hop G, G floods it throughout its GoI and, in analogy to the object insertion process (see Section 4.5), forwards it for further flooding to all its children if it has any below the current routing position. For the forwarding, G augments the position(ℓ,d) indicator in the message so that its children do not in vain try to match the same position again (which would lead to an infinite loop of messages). Every node storing links to an object matching M^0 returns them in a `query_answer` message.

Figure 11 also shows how a new node N that has not yet been fully integrated into the overlay, *i.e.*, whose time-out period τ after initial contact to SHARK has not yet exceeded (cf. Section 4.3), can query for objects. It simply sends a `query_proxy` message to its first contact A who subsequently acts as requestor for the query. A subsequently forwards all answers to N.

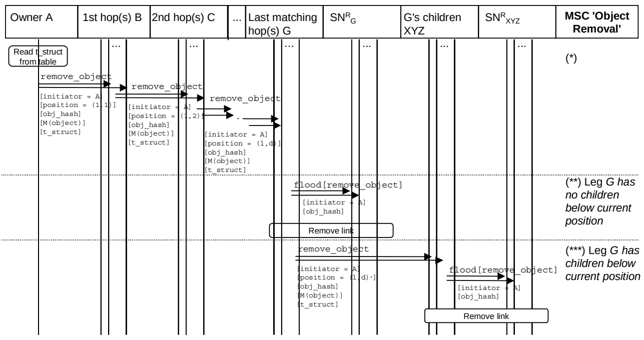


Figure 10: MSC Object Removal

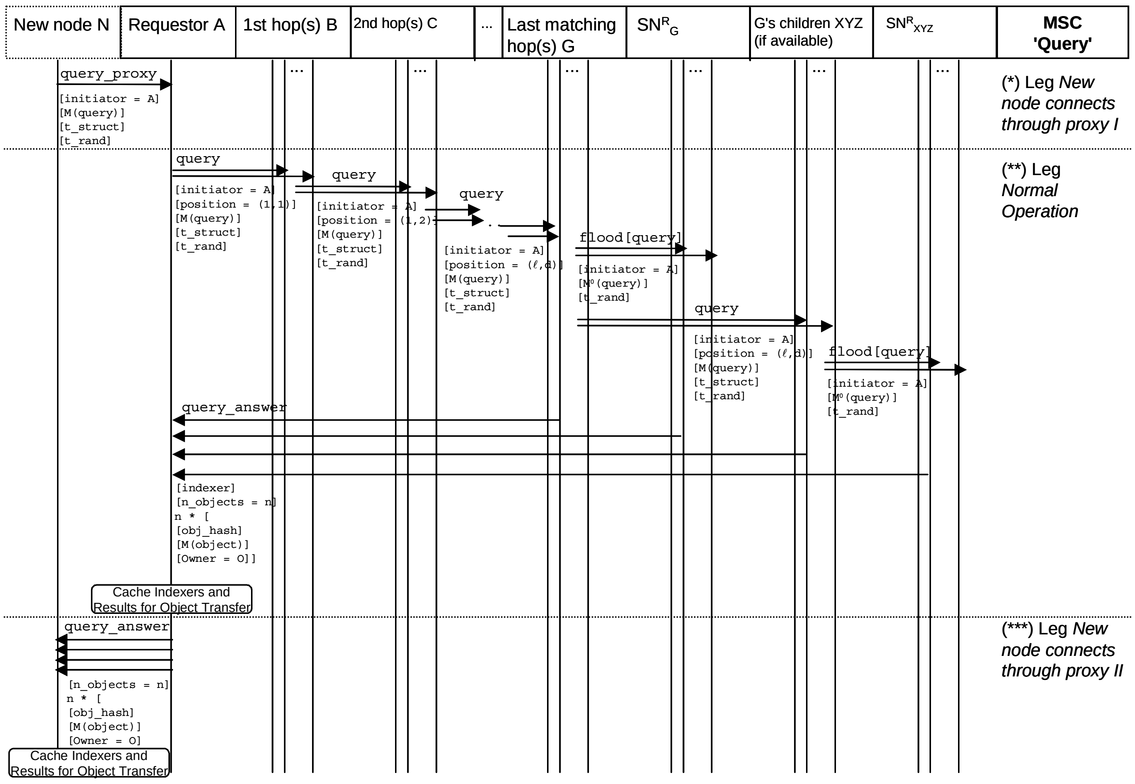
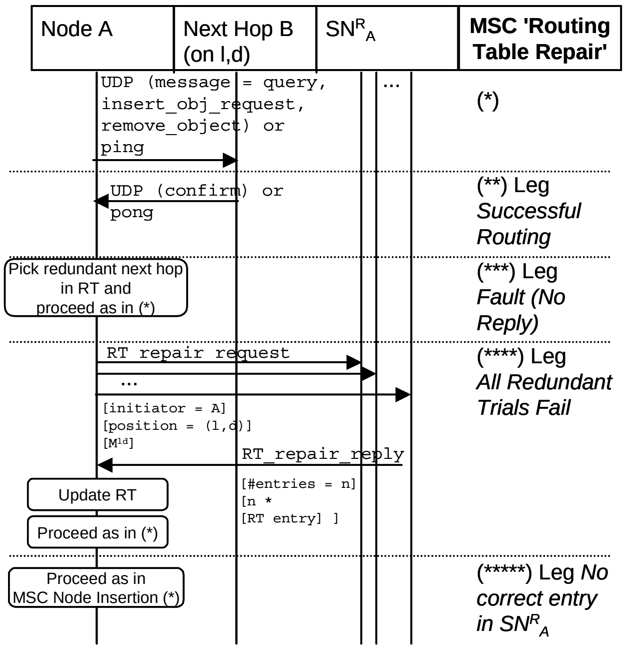


Figure 11: MSC Query

4.8 Routing Table Repair

Faults may occur when routing messages through the SHARK overlay. We assume that the network communication applied confirms receipt of a message so that faults can easily be detected by the absence of such confirmation. For instance, UDP could be used as a lightweight protocol for message exchange, but confirmation messages would have to be returned for fault detection.

Figure 12 shows how SHARK proceeds when a fault occurs. As a first step, the sender node A simply picks the



Node A
Next Hop B (on l,d)
SN^R_A
MSC 'Routing Table Repair'
UDP (message = query, insert_obj_request, remove_object) or ping
(*)
UDP (confirm) or pong
(**) Leg Successful Routing
Pick redundant next hop in RT and proceed as in (*)
(***) Leg Fault (No Reply)
RT repair request
...
[initiator = A]
[position = (l,d)]
[M^ld]
(****) Leg All Redundant Trials Fail
RT_repair_reply
[#entries = n]
[n * [RT entry]]
Update RT
Proceed as in (*)
Proceed as in MSC Node Insertion (*)
(*****) Leg No correct entry in SN^R_A

Figure 12: MSC Routing Table Repair

redundant entry for the next hop from its RT and re-transmits the message. Should the redundant entry fail as well, A sends a `RT_repair_request` to its GoI neighbors SN^R_A. As all member of a GoI have equivalent routing tables, all of them can offer replacements for the faulty nodes via an `RT_repair_reply` message. A integrates the replacements into its RT and tries sending again. Should all offered alternative entries fail as well, A re-inserts itself into the SHARK overlay as described in the MSC node insertion, thus establishing an entirely new routing table.

4.9 Object Transfer

Once a requestor A has found a desired object, he usually wants to have it transferred. For that purpose A sends a `request_object` message directly to the object owner, specifying the desired object by its object hash (see Figure 13). The owner returns a message which includes an object handle. It is up to the application using SHARK to define what this handle is and how to interpret it. For short information objects, *e.g.*, it could be the object itself, whereas for long data files, it may be a detailed download instruction. If the owner does not or no longer offer the desired object, it sets an error flag. Requestor A then has the faulty link removed at the indexer through a `report_stale_link` message.

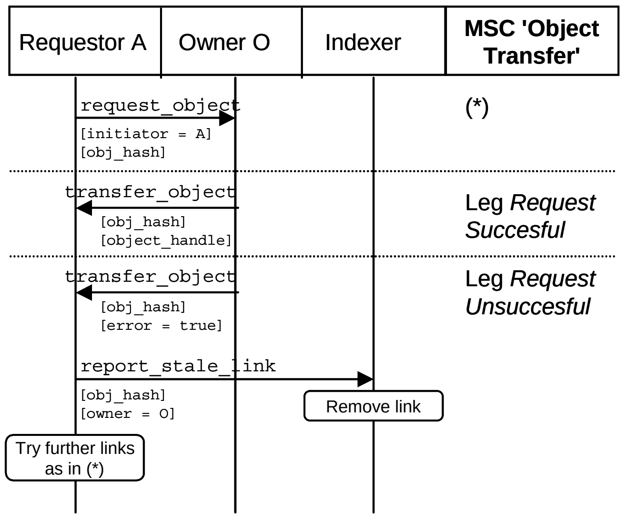


Figure 13: MSC Object Transfer

4.10 GoI split

As more and more nodes join the network, they fill and grow the existing GoIs. In order to avoid too large GoIs that limit scalability, SHARK measures GoI sizes and actively triggers nodes to split large GoIs into several smaller ones by using an additional level of the hierarchy or the meta-data descriptions, respectively. Figure 14 shows the corresponding MSC.

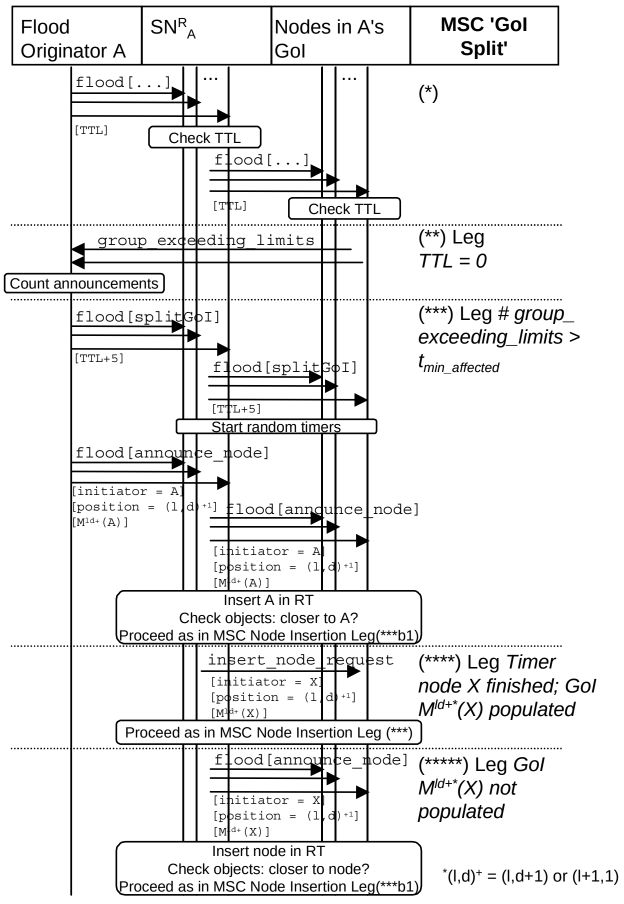


Figure 14: MSC GoI Split

Whenever a message is flooded throughout a GoI on a current leaf position(ℓ,d) in the hierarchy, starting at a node A, a time to live (TTL) field in the flood[...] message is decreased by one at every instance of forwarding. When a node receives a message with TTL=0, it sends a `group_exceeding_limits` announcement to A. Node A counts these announcements; when their number exceeds a 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 sending `group_exceeding_limits` announcements to avoid the message at subsequent flooding events. A then consults its own meta-data description to determine its position on the next lower level in the hierarchy (ℓ+1,d) (or (ℓ,d+1)) that was previously entirely unoccupied. A becomes the first member of that GoI and informs its former neighbors by flooding an `announce_node` message. The neighbor nodes update their RT with A's new position and check whether the object links they store need to be moved to A just as in the MSC Node Insertion, leg (***b1). They also start random timers. Once a timer has finished at a node B, B determines its new position on the next lower level. If the corresponding GoI has already a member (*e.g.*, A), it initiates an `insert_node_request`, and is inserted into its new GoI like in the MSC Node Insertion, leg (***). Should B's target GoI not yet be populated, it proceeds just like A before by becoming the first member and informing of this fact through an `announce_node` message. Once all timers have finished, all nodes of the split GoI have moved to their new positions one level further down the hierarchy.

4.11 GoI Insertion

The GoI split mechanism explained in the previous section requires that the meta-data hierarchy and the corresponding descriptions of nodes and objects extend in their granularity beyond what is currently actually used in the overlay structure and for routing.

Alternatively or in addition, the categorisation itself can adapt over time. A node A can create a new category GoI_{new} with an `insert_GoI_request` as indicated in Figure 15. A automatically becomes the first member of GoI_{new}. The routing procedure for `insert_GoI_request` is identical to the node insertion mechanism. After successful creation of GoI_{new}, A sends an `announce_GoI` to all sibling GoIs, *i.e.*, the children of the last routing hop, or to the parent GoI if no siblings yet exist. 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. Recipients of this message update their meta-data hierarchy and routing tables. They also ask their users if they want to be part of the new GoI and, if so, start the required node insertion. For the object links they hold, they contact the object owners and ask them about reclassification of the objects into GoI_{new}. If the owner's answer is positive, the corresponding object is transferred.

Potential requestors to GoI_{new} learn about its existence through an extended `announce_GoI_all` 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 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 from one level above the last one on downward by using wildcards for all lower levels and dimensions in the routing mechanism. All other creator nodes whose new GoI has been announced this way clear their caches. A new timer starts, and after it expires, another aggregate `announce_GoI_all` message is flooded from one more level higher on downward. This process continues up to the root of the overlay network when all nodes are informed of all recent updates. Nodes receiving an `announce_GoI_all` message update their meta-data hierarchies accordingly.

A flooding of the `announce_GoI_all` message could 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.

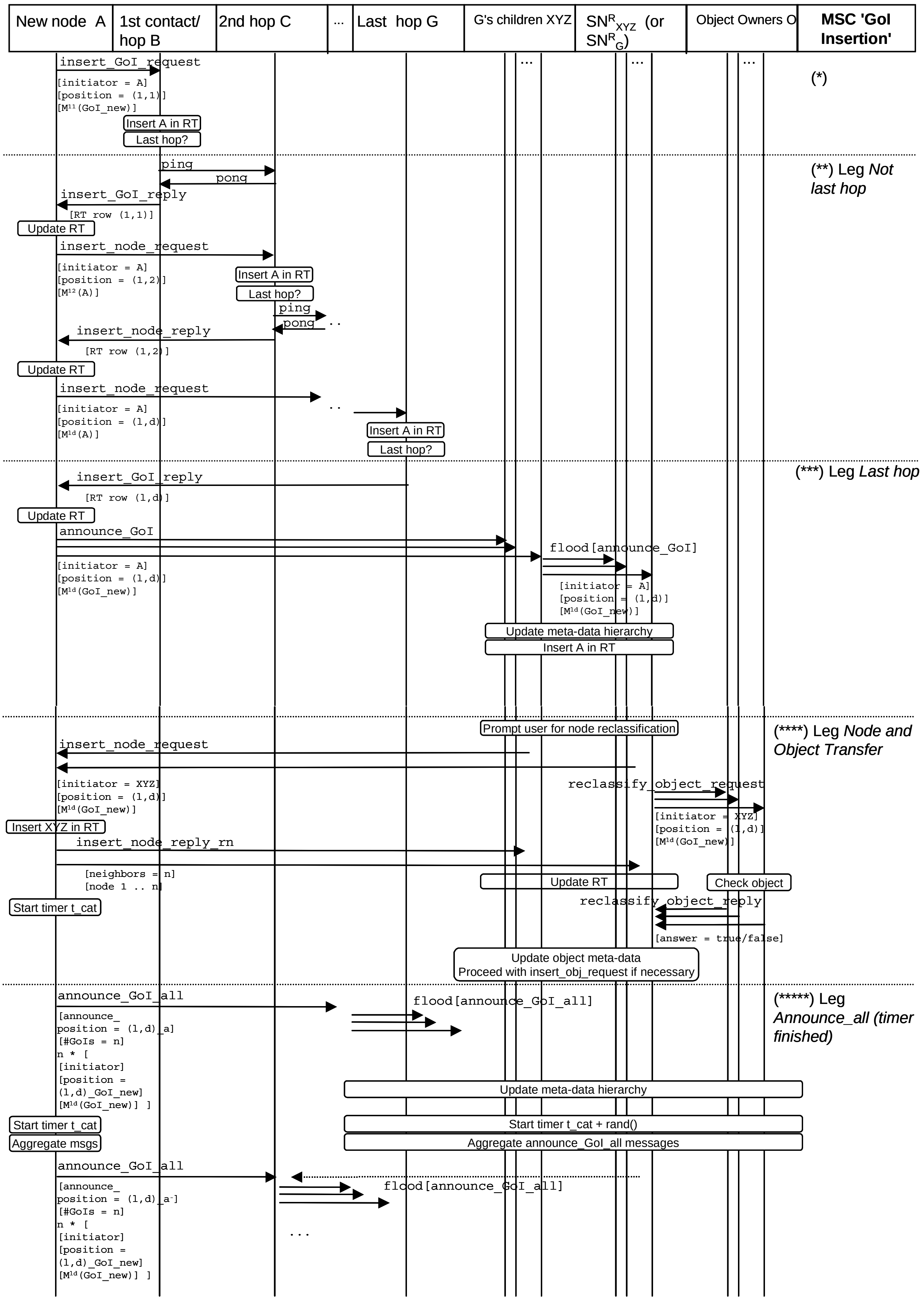


Figure 15: MSC GoI Insertion

4.12 Message Types, Fields, and Formats

Table 1 summarizes the message types used in the SHARK MSCs. Along with the message types and their encoding, the table gives a brief description of the message usage and specifies the issuer of the message, its final destination, and the way it is delivered to its destination.

Table 1: Message Types

Enco ding	Type	Description	Issuer	Path	Destination
0x10	insert_node_ request	Insert new (real or virtual) node into the SHARK overlay	User/ servent	routed	-
0x11	insert_node_ reply	Transmit next hop and part of routing table	Contacted node	1-to-1	Joining node
0x12	insert_node_reply_rn	Transmit neighbors in random network neighborhood	Contacted node	1-to-1	Joining node
0x13	announce_node	Announce existence of node in previously unoccupied GoI	New node A	1-to-many	A's siblings
0x14	remove_node	Notify neighbors of departure	Leaving node A	1-to-many	A's nei-ghbors
0x20	insert_obj_req	Insert new object link on indexing nodes	Object owner	routed	r nodes in GoI
0x21	insert_obj_reply	Confirm object insertion in GoI	First indexer	1-to-1	Owner
0x22	replicate_link	Replicate object link within GoI or transfer link	First indexer	1-to-many	r nodes in GoI
0x23	remove_object	Remove object links from indexing nodes	Object owner	routed	Indexing node
0x30	query	Route query to corresponding GoI(s)	Requestor	routed	Node in GoI
0x32	query_answer	Return links of matching objects	Indexing node	1-to-1	Re-questor
0x33	query_proxy	Send query through proxy	Reques-tor	1-to-1	Proxy
0x40	insert_GoI_ request	Route node establishing a new GoI to target destination	Inserting node	routed	-
0x41	insert_GoI_ reply	Transmit next hop and part of routing table	Contacted node	1-to-1	Inserting node
0x42	announce_GoI	Locally announce new GoI	GoI inserting node A	1-to-many	A's siblings/ par-ent
0x43	announce_GoI_all	Globally announce new GoI(s) with caching	Any node (timer)	routed	All children GoIs
0x40	reclassify_object_request	Prompt for object reclassification after GoI split/insertion	Indexing node	1-to-1	Object owner
0x41	reclassify_object_answer	Reclassify objects after GoI split or insertion	Ohject owner	1-to-1	Indexing node
0x50	group_exceeding_limits	Notify of large GoI size	Any node in GoI	1-to-1	Node A
0x60	flood[query]	Flood query in GoI	Node A	flood	GoI
0x61	flood[announce_node]	Flood announce_node in GoI	Node A	flood	GoI
0x62	flood[remove_node]	Flood remove_node in GoI	Node A	flood	GoI
0x63	flood[remove_object]	Flood remove_object in GoI	Node A	flood	GoI
0x64	flood[splitGoI]	Flood notification that GoI should split	Node A	flood	GoI
0x65	flood[announce_GoI]	Flood announce_GoI in GoI	Node A	flood	GoI
0x66	flood[announce_GoI_all]	Flood announce_GoI_all in GoI	Node A	flood	GoI
0x72	RT_repair_request	Request update for stale routing table entry	Repairing node A	1-to-many	A's neighbors
0x73	RT_repair_reply	Transmit routing table entries to requestor for repair	A's neighbor	1-to-1	Node A
0x80	request_object	Request transfer of found obj.	Request.	1-to-1	Owner
0x81	transfer_object	Transfer object or object handle to requestor	Owner	1-to-1	Requestor
0x82	report_stale_link	Notify indexer of stale link	Requ-estor	1-to-1	Indexer
-	ping	ICMP echo request to test if neighbor is alive	Routing node	1-to-1	Next hop
-	pong	ICMP echo reply to confirm being alive	ping recipient	1-to-1	Ping originator

Table 2 presents the lengths and body fields for the different SHARK message types. The version number is currently obviously 1 for all messages. The body length is the sum of the lengths of each field contained in the message. All messages requiring flooding have been aggregated in one part of the table as the same basic mechanism for flooding can be used for all of them.

Table 2: Messages

Ver.	Type	Body Length [Byte]	Body fields
1	insert_node_ request	6 + 2 + variable	initiator | position(ℓ,d) | meta_data_pos $M^{\ell d}$
1	insert_node_ reply	variable	routing_table_row
1	insert_node_reply_rn	n * 6	#neighbors | rt_entry1 | rt_entry2 ...
1	announce_node	6 + 2 + variable	initiator | position(ℓ,d) | meta_data_pos $M^{\ell d}$
1	remove_node	n * 6	#entries | rt_entry1 | rt_entry2 ...
1	insert_obj_req	variable	initiator | position(ℓ,d) | obj_hash | meta_data | t_struct | replication

Table 2: Messages

Ver.	Type	Body Length [Byte]	Body fields
1	insert_obj_reply	8 + 6 + variable	initiator | meta_data
1	replicate_link	variable	initiator | obj_hash | meta_data | replication
1	remove_object	variable	position(ℓ,d) | obj_hash | meta_data | t_struct
1	query	variable	initiator | position(ℓ,d) | meta_data | t_struct | t_rand
1	query_answer	variable	indexer | n_objects | n * [obj_hash | meta_data | Owner]
1	query_proxy	variable	initiator | position(ℓ,d) | meta_data | t_struct | t_rand
1	insert_GoI_ request	6 + 2 + variable	initiator | position(ℓ,d) | meta_data_descr $M^{\ell d}$
1	insert_GoI_ reply	variable	routing_table_row
1	announce_GoI	6 + 2 + variable	initiator | position(ℓ,d) | meta_data_descr $M^{\ell d}$
1	announce_GoI_all	2 + 2 + n * var.	announce_position(ℓ,d) | #GoIs | n * [initiator | position (ℓ,d) | meta_data $M^{\ell d}$]
1	reclassify_ object_request	6 + 2 + variable	initiator | position(ℓ,d) | meta_data_descr $M^{\ell d}$
1	reclassify_ object_answer	1	answer
1	group_exceeding_limits	0	-
1	flood[query]	1+8+6+var.+1	TTL | msg_ID | initiator | meta_data M^0 | t_rand
1	flood[announce_node]	1+6+2+variable	TTL | initiator | position(ℓ,d) | meta_data_pos $M^{\ell d}$
1	flood[remove_node]	1+6	TTL | initiator
1	flood[remove_object]	6+16	initiator | obj_hash
1	flood[splitGoI]	1	TTL
1	flood[announce_GoI]	6 + 2 + variable	initiator | position(ℓ,d) | meta_data_descr $M^{\ell d}$
1	flood[announce_GoI_all]	2 + 2 + n * var.	announce_position(ℓ,d) | #GoIs | n * [initiator | position (ℓ,d) | meta_data_descr $M^{\ell d}$]
1	RT_repair_request	6 + 2 + variable	initiator | position(ℓ,d) | meta_data_pos $M^{\ell d}$
1	RT_repair_reply	n * 6	#entries | n * [rt_entry]
1	request_object	6 + 16	initiator | obj_hash
1	transfer_object	variable	obj_hash | flag_error | object_handle
1	report_stale_link	16 + 6	obj_hash | owner

Table 3 explains the body fields contained in the SHARK messages in more detail, including their lengths in Byte, their formats, and brief descriptions. Some fields require some more explanation, particularly the meta-data. Assuming the meta-data hierarchy has been defined by the application developer in sufficient depth and breadth, its structure and entries are known to all peers. Hence, a user could easily select exact categories for his requests from lists or menus, rather than trying his own descriptions. In this case, SHARK can translate the category descriptions into numeric index positions in the hierarchy and use these positions rather than string expressions for routing, reducing the processing load for routing and the message sizes. This dual way of specify-

Table 3: Message Fields

Field	Length [Byte]	Format	Description
initiator, owner, indexer	6	ipv4:port	IP address and port of peer that initiated the message (or owns/indexes object)
position(ℓ,d)	1 + 1	Byte	Position (dimension d on level ℓ) last resolved by routing engine on forwarding peer
announce_position(ℓ,d)	1+1	Byte	Position from which downward announce_GoI_all message is currently being flooded
#neighbors	1	Byte	Number of neighbors being sent in RT_repair_reply or insert_node_reply_rn
#GoIs	2	Integer	Number of new GoIs being announced in announce_GoI_all message
n_object	2	Integer	Number of object links in query_answer
rt_entry	6	ipv4:port	IP address and port of routing table entry
obj_hash	16	Integer	MD5 hash of object serving as a unique identifier of that object
t_struct	1	Byte	t_struct/255*100%: Minimum required match of query with meta_data description of potential next hop in hierarchy, so that request is forwarded
t_rand	1	Byte	t_rand/255*100%: Minimum required match of query meta_data_M0 (see below) with meta_data description M0 of object so that object link is returned to requestor
replication	1	Byte	Number of desired replicas when inserting an object link for redundancy / fault tolerance

Table 3: Message Fields

Field	Length [Byte]	Format	Description
answer	1	Boolean	Object in newly inserted GoI? (yes/no)
TTL	1	Byte	Time to live counter in flooding messages
flag_error	1	Byte	Indication that object not available on owner
object_handle	variable	applicat.-def.	Actual object found and to be transferred, or respective object handle, opaque to SHARK
meta_data M^0	2 + variable	Integer: String	Keyword (or keyword expression with wildcards, AND, OR) describing an object within its GoI; preceeded by 2-Byte length indication
meta_data_pos $M^{\ell d}$	2	Integer	Index number of meta-data (of query or object or node) on position(ℓ,d)
meta_data_descr $M^{\ell d}$	2+2+ variable	Index(Int.): Length(Int.): Descr.(String)	Description of new GoI, together with length indication of the description and new index number in RT on position(ℓ,d)
meta_data (opt1)	1+ 2+var.+2 + n*(2+2)	see right	flag_opt1=true | meta_data M^0 | #entries | n * [position(an,bn) | meta_data_pos M^{anbn}] Meta-data of an object/request/node in terms of the succession of index positions of its description
meta_data (opt2)	1+2+var. 2+ n* (2+var)	see right	flag_opt1=false | meta_data M^0 |#entries| n * [position(an,bn) | meta_data_descr $M^{an,bn}$] Meta-data of an object/request/node in terms of the succession of meta-data description on each level/dimension
routing_table_row	2 + n*(2+6)	see right	#entries | n * [meta_data_pos | rt_entry] Routing table row with total number of entries, for each entry the meta_data index position and the routing table entry

ing categories is reflected in the fields meta_data_pos (index-based specification) and meta_data_descr (string-based specification), where the latter one obviously has to be used for new GoI insertion. Just like the categorisation on each hierarchy level and dimension, the same dual description mode can also be applied to the entire classification path from the root to the leaves: meta_data(opt1) is based on index usage whereas meta_data(opt2) is based on string descriptions. SHARK is designed to correctly handle both of them. Note that the description M^0 within a GoI obviously always has to be a keyword expression.

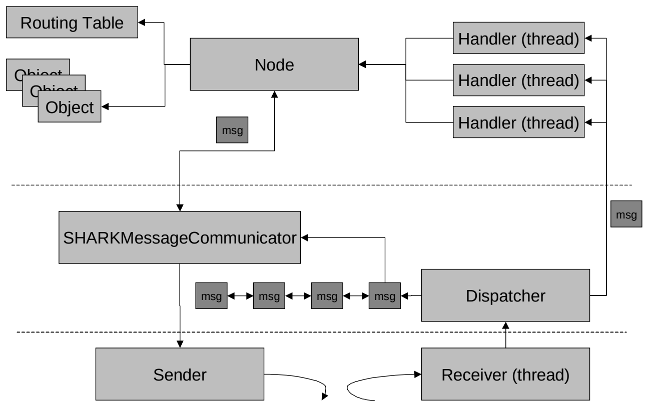


Figure 16: Implementation Structure

5 Implementation

SHARK has been implemented in [21] on Windows XP in Java such as to be easily portable to other platforms. As a proof of concept, the software implements the overall architecture, the communication infrastructure, a correct handling and dispatching of all message types, and the use cases node insertion, object insertion, and query.

Figure 16 illustrates the module structure. To allow for an easy exchange of communication mechanisms, a ’SHARK Node’ passes all messages to the ’SHARK 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.

6 Evaluation

We provide a detailed evaluation of SHARK in this section. After taking a closer look at the hierarchy structure and its dependence on network size and meta-data categories in Section 6.1, we investigate the scalability of queries in Section 6.2. We then evaluate the cost of overlay network management in Section 6.3 before summing up in Section 6.4.

6.1 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. Particularly relevant for subsequent subsections are the number levels, the degree of imbalance, and the occupation of GoIs.

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 many times (cf., *e.g.*, [11]) 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 [28] of P2P networks like Gnutella suggest a bi-modal Zipf distribution with constant (α_1=0) popularity up to an inflection point. Further studies experiment with exponential distributions but do not parameterize on the network size [29]. We choose α_2=1, which corresponds to the original version in [11] and is the average reported in [28], and we set inflection at t_{inf}=1% of possible GoI ranks. With the number of GoIs or categories n_{cat}, the relative GoI popularity becomes

$$p(i) = \begin{cases} c\dfrac{\lfloor n_{cat}t_{inf}\rfloor^{\alpha_2-\alpha_1}}{i^{\alpha_2}}; & i > \lfloor n_{cat}t_{inf}\rfloor \\ \dfrac{c}{i^{\alpha_1}}; & i > \lfloor n_{cat}t_{inf}\rfloor \end{cases};$$

$$c = \left[\sum_{i=\lfloor n_{cat}t_{inf}\rfloor}^{n_{cat}} \frac{1}{i^{\alpha_1}} + \lfloor n_{cat}t_{inf}\rfloor^{\alpha_2-\alpha_1} \sum_{i=1}^{\lfloor n_{cat}t_{inf}\rfloor} \frac{1}{i^{\alpha_2}}\right]^{-1}$$

Figure 17 evaluates the effect of node distribution to GoIs on the SHARK hierarchy. The dashed line shows the average

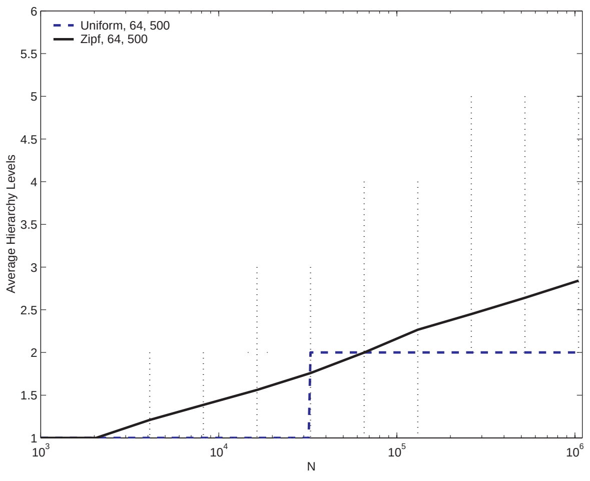


Figure 17: Hierarchy Levels

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 well below three even at a million nodes and increases only logarithmically.

The choice of meta-data structure, most notably the number of categories per level, is evaluated in Figure 18, assuming a bi-modal Zipf distribution for nodes to GoIs and comparing maximum GoI sizes of 100 and 500 nodes, respectively. It is obvious that the number of levels falls with

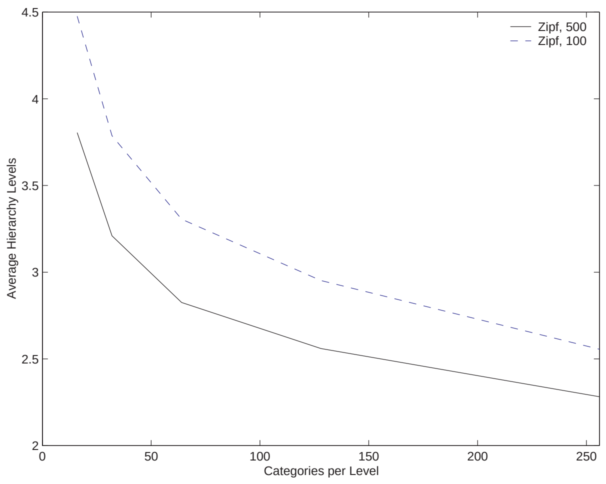


Figure 18: Hierarchy Levels

growing GoI size and with growing number of categories per level. In the following, we will always assume a maximum GoI size of 500 nodes and 64 meta-data categories per level unless otherwise stated.

The final issue we are interested in regarding the SHARK hierarchy is the number of GoIs that are available yet remain empty due to a low popularity. This has a significant effect on node insertion cost due to the additionally required announcements (cf. Section 4.3). Figure 19 shows the fraction of unoccupied GoIs for both, a maximum GoI size of 500 and of 100 (solid and dotted lines, respectively). Both are identical for the first few hundred nodes during the filling of the first hierarchy level with nodes. While some 10% of GoIs subsequently remain empty for the smaller GoI size due to ongoing group splitting, the ratio of empty GoIs is identical zero for the larger group size: a sufficient number of nodes is statistically available upon splitting to provide members even for the less popular groups. Figure 19 also presents the even more important ratio of node insertion, object insertion, or query requests that are targeted to empty GoIs (dashed and dash-dotted lines, respectively). As empty GoIs are also the least popular ones, this ratio is identical zero or close to zero for all network sizes larger than a few hundred nodes.

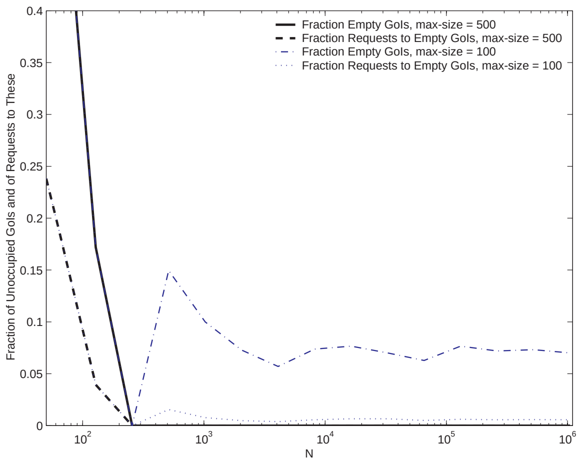


Figure 19: Ratio of Unoccupied GoIs

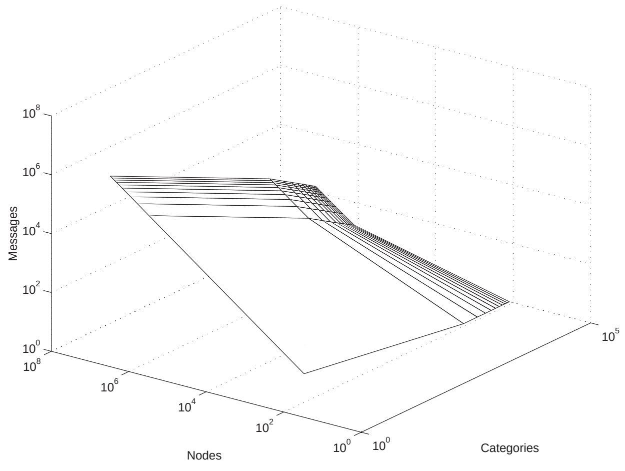


Figure 20: Number of Messages for a Query

6.2 Cost of Queries

For the cost of queries, we consider the aggregate number of messages necessary to resolve a query as it determines bandwidth needs, and the request pathlength as the key driver of latency. We also evaluate the node degree due to its relevance for processing power and memory demand. For completeness, the message size can be obtained from Table 2.

6.2.1 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]. With the pruning probability p_p as in Section 3.4 and the definition of n_{step} as the maximum number of total steps in the hierarchy, *i.e.*, in the balanced case, the number of levels times the number of dimensions on each level, h_h becomes

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

Figure 20 visualises the number of messages in SHARK, assuming a bi-modal Zipf distribution of queries and nodes to categories as usual, 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. m_{agg} significantly drops with the number of categories and grows sub-linear when the number of categories is increased with additional nodes.

Figure 21 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, and that the Gnutella reference case has a sufficiently large time to live. In small networks, while the GoIs are being filled, SHARK achieves roughly an order of magnitude improvement over Gnutella. Once considerable GoI splitting occurs, the bandwidth demand in SHARK grows sub-logarithmic rather than linear as in Gnutella. This represents the key achievement of SHARK.

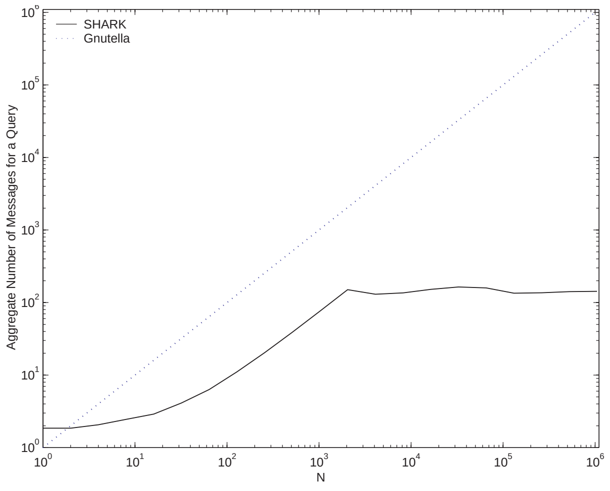


Figure 21: Aggregate Messages for a Query

6.2.2 Request Pathlength

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. [23] shows that node connectivity in a Gnutella network (*i.e.*, random network), follows a bi-modal power law distribution, where the probability p_ℓ that a node has i links remains constant up to 10 links:

$$p_\ell(i) = \begin{cases} \dfrac{c_1}{10^{\alpha_1}}; & i \le 10 \\ \dfrac{c_1}{i^{\alpha_1}}; & i > 10 \end{cases}$$

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.

where α_1 and c_1 can be determined from $\sum_{i=1}^{n} p_\ell(i) = 1$ and $\sum_{i=1}^{n} i p_\ell(i) = \bar{\ell}$, and $\bar{\ell}$ is the average number of links. The average number of hops within a GoI becomes

$$\bar{h}_{GoI} = \sum_{i=1}^{n(GoI)} p_\ell(i)\left[\frac{i}{n(GoI)} + \sum_{j=1}^{n(GoI)-1} p(n_2 = j \mid n_1 = i) \cdot \left[2\frac{j}{n(GoI)} + \sum^{n(GoI)-i-j} p(n_3 = k \mid n_2 = j)\left[3\frac{k}{n(GoI)} + \ldots\right.\right.\right.$$

where

$$p(n_h = i \mid n_{h-1} = k) = \sum_{\nu_j} \prod_{j=1}^{k} p(\nu_j);\ \nu_j \in \aleph;\ \sum_{j=1}^{k} \nu_j = k + i$$

is the conditional probability that i additional nodes are contacted at hop h given that k nodes have been contacted at the previous hop h-1. This is equivalent to the probability that k nodes have a total of k+i connections (i additional contacted nodes, k backward links to the nodes of the previous hop). Each node j contacted at hop h-1 has ν_j connections with probability $p_\ell(\nu_j)$; the product of the individual probabilities yields the probability of the overall constellation.

The calculation can be vastly simplified assuming a constant average node connectivity $\bar{\ell}$:

$$\bar{h}_{GoI} = \sum_{h=1}^{h_{max}-1} h\frac{n_{incr}(h)}{n(GoI)} + h_{max}\frac{n(GoI) - \sum_{i=1}^{h_{max}-1} n_{incr}(h)}{n(GoI)}$$

where $n_{incr}(h)$ is the incremental number of nodes contacted at hop h

$$n_{incr}(h) = \begin{cases} 1; & h = 0 \\ \bar{\ell}; & h = 1 \\ \bar{\ell}(\bar{\ell}-1)^{h-1}; & h > 1 \end{cases}$$

h_{max} is the maximum number of hops in a GoI and can be derived from

$$\sum_{h=1}^{h_{max}-1} n_{incr}(h) < n(GoI),\quad \sum_{h=1}^{h_{max}} n_{incr}(h) \geq n(GoI)$$

The request pathlength PL_R within the entire SHARK is then simply the number of hops in the hierarchy $\bar{h}_h$ 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 relative request frequencies to GoIs so that the larger, more popular GoIs with unfortunately more hops are queried more frequently.

Figure 22 compares the request pathlengths of SHARK (solid line) and Gnutella (dotted line), with the same assumptions as for m_{agg} and based on a pruning probability p_p=5% and an average node connectivity $\bar{\ell} = 3.4$. It is shown that SHARK may even outperform the already benign logarithmic behaviour of Gnutella.

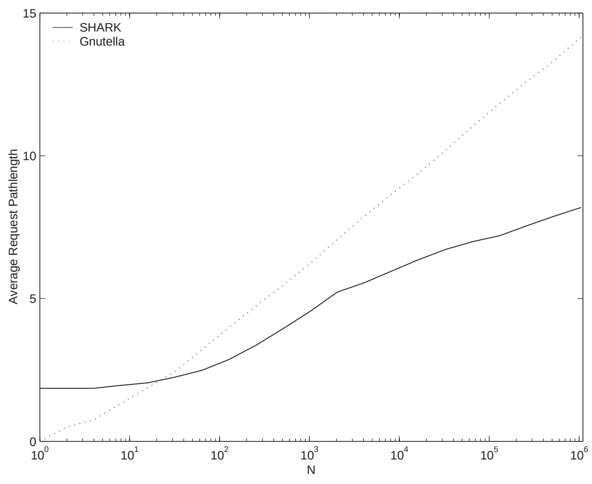


Figure 22: Request Pathlength

6.2.3 Node Degree

Figure 23 depicts the average node degree for a SHARK node and is also representative of its routing table size. The

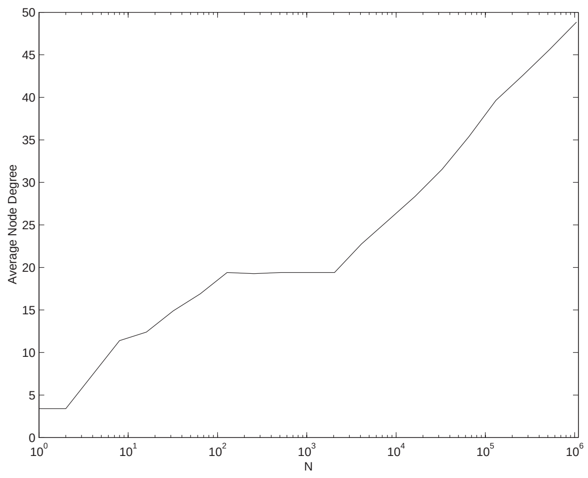


Figure 23: Node Degree

node degree accounts for 8 nodes on each level and in each dimension plus $\bar{\ell} = 3.4$ neighbors in the random network part. The number of levels is determined as in Section 6.1 and averaged across all nodes. The node degree 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 networks grows further and GoI splitting occurs. The node degree remains below 50 at a million nodes.

6.3 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.

6.3.1 Cost of Node Insertion

The cost, *i.e.*, the aggregate number of messages to insert a node, is depicted in Figure 24. It accounts for an `insert_node_request`, a `ping`, a `pong`, and an `insert_node_reply` message to be sent at each hop as well as for the final $\bar{\ell}$ `insert_node_request` and $\bar{\ell}+1$ `insert_node_reply_rn` messages. For reasonably large networks, the cost develops logarithmically up to some 35 messages in systems of one million nodes. A significantly higher cost is incurred in young networks as they grow toward the first GoI splitting. As described in Section 4.3, when a new node is the first member of a previously empty GoI (cf. Section 6.1), 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. For the transfer probability, we considered the average relative popularity of empty groups within the parent GoI.

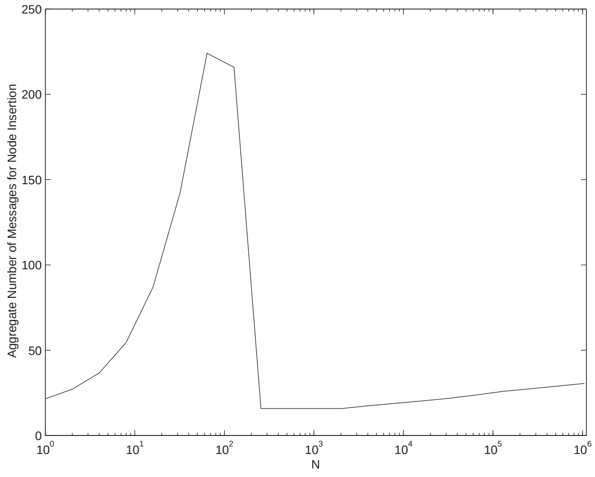


Figure 24: Cost of Node Insertion

6.3.2 Cost of Node Removal

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 25. 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.

6.3.3 Cost of Object Insertion

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 and create redundancy. Figure 26 depicts the total assuming a

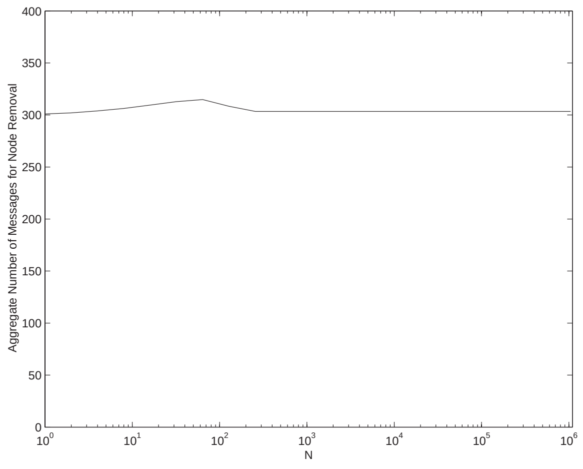


Figure 25: Cost of Node Removal

pruning probability p_p=5% as before. It grows logarithmically with the size of the network and remains very low compared to the cost of queries.

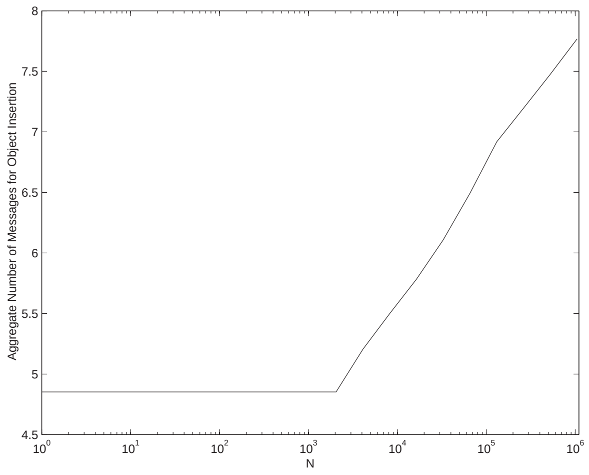


Figure 26: Cost of Object Insertion

6.3.4 Cost of Object Removal

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.

6.3.5 Cost of GoI splitting

While being irrelevant in a stable system state, 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 of the n(GoI) nodes in the group. For each of the s (s=64) new subgroups, one node will become its first member and announce its presence to each of the n(GoI)-1 other nodes. In order to reach its new target subgroup, each node follows the hierarchy one level further down through an `insert_node_request`, entailing a `ping`, a `pong`, and an `insert_node_reply` message, and, finally, additional `insert_node_request` and `insert_node_reply_rn` messages to insert the node into its new GoI. Finally, each node has to transfer each of its links with a transfer probability

$$p_{\text{transfer}} = \sum_{i=1}^{s} p_{\text{Zipf}}(i)(1 - p_{\text{Zipf}}(i))$$

where the first factor is the percentage of object links that moved to a certain subgroup and the second factor the probability that these links in fact belong to any one of the other subgroups. Overall, the number of messages for a GoI split becomes

$$n_{\text{GoI\_Split}} = n(\text{GoI})[s + 7 + p_{\text{transfer}} r|o|] - s$$

and is depicted in Figure 27 as a function of the maximum GoI size. The figure also shows the cost for s=16 subgroups upon splitting for comparison; clearly, a higher number of subgroups leads to increased traffic due to node announcements.

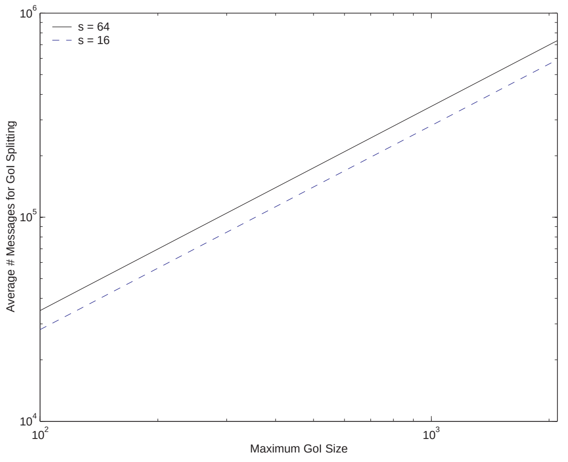


Figure 27: Cost of GoI Splitting

6.3.6 State Information

In addition to an evaluation of the number of messages for different overlay management use cases as in the previous paragraphs, it is also important to have a look at the information to be kept in memory on each node. This information 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 numerical categorisation into the meta-data structure `meta_data_pos`, and a keyword description M^0. The meta-data hierarchy requires a description to be stored for each GoI and each higher-level category. Note that storage of the meta-data hierarchy could be avoided when routing directly according to hierarchical keyword descriptions `meta_data_descr` rather than their numerical representations `meta_data_pos`. The total state information per node is plotted in Figure 28, 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

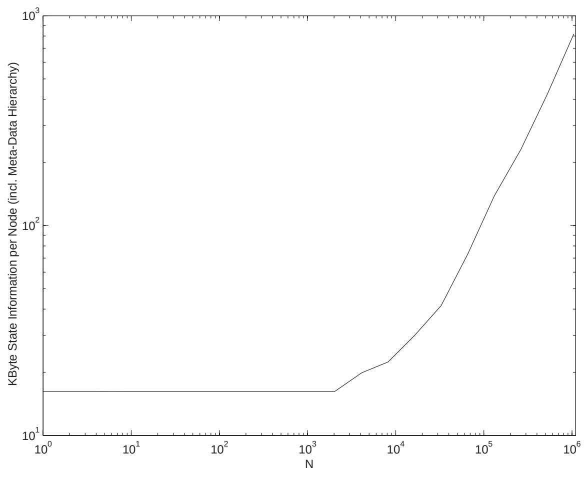


Figure 28: State Information per Node

description. It remains well below 1 MByte for networks as large as 1 million nodes.

6.4 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 the SHARK search system. Figure 29 compares the aggregate number of messages per node and day or second of Gnutella and SHARK in three 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 recognise 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 with only roughly one message per minute in a network of one million nodes. It also demonstrates logarithmic or even sub-logarithmic scalability as long as the meta-data hierarchy can be defined and applied in sufficient depth.

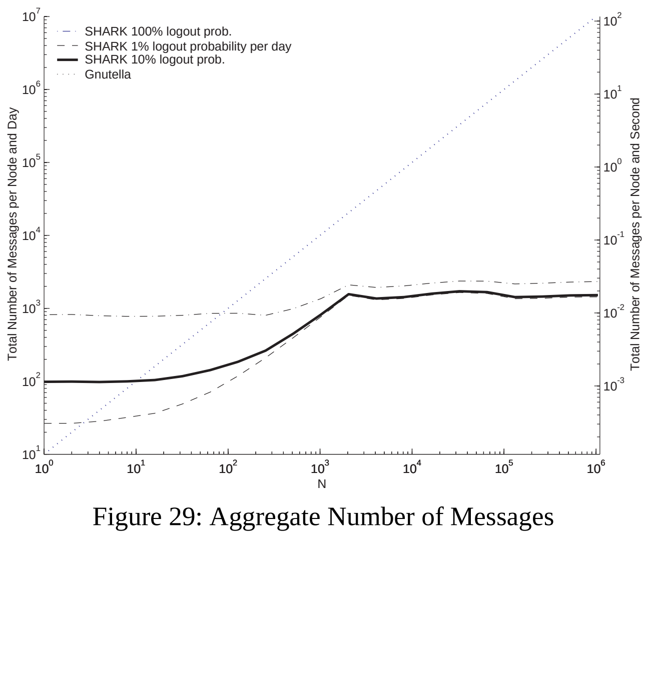


Figure 29: Aggregate Number of Messages

7 Conclusion

In this document, we have specified a novel approach to P2P keyword search, SHARK. It builds on a multi-dimensional meta-data structure to classify nodes and objects. A multi-dimensional symmetric redundant hierarchy of GoIs is built in accordance with the meta-data structure so as to allow efficient keyword routing. By restricting flooding to small semantic clusters or GoIs, it achieves vast performance improvements of up to four orders of magnitude compared to random networks like Gnutella.

We expect SHARK to be applied to 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 P2P trading systems, where advertisers can insert their trade goods like classified ads according to object classifications and price information. Knowledge marketplaces and expert systems usually build a more or less granular ontology of topics anyway that can straightforwardly be exploited by SHARK to create an according structure. Finally, in collaboration tools, users may create their own structures for document classification, similarly to standard file systems. These structures could also be matched by the SHARK topology and could be updated at the users' discretion.

Going forward, we would like to build a P2P trading application on top of SHARK. Furthermore, a detailed investigation of security issues relating to SHARK or any other structured approach to P2P lookup and search would be helpful. Denial of service attacks are a particular threat.

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] J.J. Aucouturier, F. Pachet: *Representing Musical Genre: A State of Art*; To Appear, Journal of New Music Research.

[2] 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.

[3] M. Balazinska, H. Balakrishnan, D.

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

[5] M. Castro, P. Druschel, Y. C. Hu, A. Rowstron: *Exploiting network proximity in peer-to-peer overlay networks;* Future Directions in Distributed Computing (FuDiCo), Bertinoro, Italy, June 2002.

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

[7] 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.

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

[9] V. Darlagiannis: *Search in P2P Networks*; Talk, Ipswich, UK, December 2002.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] 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.

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

[20] 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.

[21] D. Reichle: *Search in Peer-to-Peer Networks*; Semester Thesis TIK SA2003-34, ETH Zurich, Switzerland, July 2003.

[22] 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.

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

[24] 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.

[25] S. Saroiu, P. Gummadi, S. Gribble: *Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts*; Multimedia Systems Journal, Vol. 8, Nr. 5, November 2002.

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

[27] 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.

[28] 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).

[29] B. Yang, H. Garcia-Molina: *Comparing Hybrid Peer-to-Peer Systems;* 27th International Conference on Very Large Databases (VLDB), Roma, Italy, September 2001.

[30] 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.

[31] 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.

[32] B. Zhao, A. Joseph, J. Kubiatowicz: *Locality Aware Mechanisms for Large-scale Networks;* International Workshop on Future Directions in Distributed Computing (FuDiCo), Bertinoro, Italy, June 2002.