PEER-TO-PEER OVERLAY NETWORK MANAGEMENT THROUGH AGILE

Jan Mischke[1] and Burkhard Stiller[1,2]
[1]ETH Zurich, Switzerland; [2]University of Federal Armed Forces, Munich, Germany

Abstract: Currently, state of the art peer-to-peer (P2P) lookup mechanisms actively create and manage a peer application layer overlay network to achieve scalability and efficiency. The proposed mechanism AGILE (Adaptive, Group-of-Interest-based Lookup Engine) extends this management approach, adapting the overlay network such as to bring requesting peers and desired lookup items close together, reducing the number of hops and, thus, latency as well as bandwidth requirements for a lookup. At the same time, AGILE introduces mechanisms to build a fair system.

Key words: Peer-to-peer (P2P) Lookup Services, Overlay Network Management, Scalability

1. INTRODUCTION

Peers in a P2P system communicate on a logical overlay network among them. Some existing systems, *e.g.*, Gnutella, build this overlay network at random, adding (or removing) links and nodes in an uncontrolled way through arbitrary ping requests and pong responds. Unfortunately, the orderless structure requires a non-scalable flooding mechanism for lookup, and the path lengths and node degrees can become large. More sophisticated approaches, like Tapestry [16], Pastry [4], or Chord [1], actively manage the overlay network such as to ensure robustness and alleviate lookup and request routing. These systems, however, pay little attention to the heterogeneity of peers with respect to their interests and capabilities.

The proposed mechanism AGILE (Adaptive, Group-of-Interest-based Lookup Engine) creates and maintains an overlay network according to specific topological requirements for P2P lookup. It additionally adapts the network over time so that groups can form according to common interests, improving the lookup performance, while at the same time ensuring fairness.

Essential topological requirements are derived in Section 2, while Section 3 discusses related work and identifies major gaps to those requirements. Section 4 introduces and evaluates the proposed approach AGILE, before Section 5 concludes.

2. REQUIREMENTS

It is straightforward to require that a P2P system be scalable and make efficient use of system and peer resources, namely *memory*, *processing power*, *bandwidth*, and *time/latency*. With up to 96% of local peer node resources being idle [2], bandwidth and user time, or latency in the technical system, are most crucial and will be considered in more detail in the next subsection. Furthermore, the system should ensure a proper *load balancing* in that it be *fair*, involving peers according to their use of the system and in that it pay attention to the *heterogeneous capabilities* of peers. Finally, a P2P system has to be *robust* to frequent node joins and leaves and link failures.

In general, network topologies can be characterized through their degree of symmetry, the network diameter, the bisection width, the average node degree, and the average wire length [6]. The functional and performance requirements (see above) determine the desired target characteristics.

- **Symmetry:** Only symmetric topologies are appropriate for true peer-to-peer systems as only in this case all peers are equal from a topology point of view. At the same time, symmetry assists load balancing. Examples of symmetric topologies include rings, buses, hypercubes, complete meshes, cube-connected circles, or k-ary n-cubes. Measurements as stated in [13], however, prove a huge heterogeneity among peer nodes in terms of their uptime, average session duration, bottleneck bandwidth, latency, and the number of services or files offered, so that server-like roles in a P2P network may be advantageous.
- **Network Diameter (D):** The diameter of a network is defined by the number of hops required to connect from one peer to the most remote peer. It strongly influences latency and bandwidth.
- **Bisection Width (β)**: The number of connections from one part of the overlay network to the other define its bisection width. Assuming proper load balancing, the maximum throughput of the network is proportional to the bisection width, and there is a direct relation with fault tolerance: the bisection width determines the number of links that have to break before the system goes down or, at least, operates only as two partial systems.
- **Node Degree (d):** The node degree is defined as the number of links that each peer has to maintain. The node degree can be a significant inhibitor for scalability. The node degree determines the size of the routing table on each peer with the proportional impact on memory consumption and processing power.
- **Wire Length (τ):** The wire length is the average round trip delay of a connection, contributing to the latency in the system.

It is particularly important to have a look in detail at latency and bandwidth consumption for a lookup request. The latency L for a lookup request is defined as

$$L = \bar{\tau} \cdot n_h = \bar{\tau} \cdot D \cdot (1 - f_p)$$

where n_h is the number of hops for a request and the *pruning factor* f_p denotes the average percentage of the maximum number of hops that a request does not need to travel, because it has been pruned off before. The pruning factor can be calculated from the pruning probability at each hop $p_{p,i}$ (i.e. the probability that the requested item is found at that hop) through

$$fp = 1 - \frac{1}{D} \sum_{i=1}^{D} \prod_{k=0}^{i-1} (1 - p_{p,k}); \qquad i, k \in \mathbb{N}$$

The pruning probability $p_{p,0}$ at node 0, the requesting node, will usually be zero. Hence, three important factors determine the latency time: the network diameter, the average round trip delay, and the pruning probability. It is possible to increase the pruning probability in a topology by exploiting knowledge on the peers' interests.

In addition, the total bandwidth B required for a lookup request is

$$B = B_{RP} n_h (d - (d-1)\varepsilon_{route}) = B_{RP} (d - (d-1)\varepsilon_{route} D(1 - f_p))$$

where B_{RP} denotes the bandwidth or size of one request package, n_h (as above) the number of hops, d the node degree, and ε_{route} the *routing efficiency*. The routing efficiency is defined to be 1 if only one node has to be contacted at each hop and 0 if all nodes have to be contacted. In that sense, Gnutella with its flooding approach has a routing efficiency of 0, whereas consistent hashing algorithms like Chord [1] have a routing efficiency of 1.

As for the latency, the network diameter and the pruning probability influence the bandwidth requirements (and scalability) in a major way. Furthermore, the routing efficiency plays a significant role. The equation also suggests that the node degree be kept low. However, this applies only if the routing efficiency is smaller than 1, as a lower node degree automatically entails a larger network diameter.

3. RELATED WORK

Tapestry [16], Pastry [4], Chord [1], and CAN [9] determine the systems most closely related to AGILE. Their common theme is that they arrange lookup items or *keys* (such as content files, services, or peer node addresses) and peer nodes in the same identifier space. Subsequently, they hand over the responsibility for holding a key with a certain identifier to a peer with a numerically close identifier. This enables them to simply route a lookup request message at each node towards a neighboring node with a closer node ID, achieving a routing efficiency of 1. All of these lookup services propose hashing to map lookup item names and nodes (IP addresses) onto the identifier space. Firstly, the hash function is globally known, ensuring the same mapping for each request for or insert of a key. Secondly, hashing results with high probability in unique IDs. Thirdly, the pseudo-randomness of the hash function uniformly distributes keys and nodes in the identifier space.

The main difference between these approaches is the topology they build to arrange peers properly so that they can route closer to the desired ID, while meeting major requirements to a good topology (cf. Section 2). Furthermore, they apply different algorithms to constructing, maintaining, or managing this topology.

- *Tapestry:* Tapestry builds a Plaxton mesh. IDs are represented as numbers with a sequence of digits to a base b. At each hop, a request is routed toward a node, whose ID matches the search key in one digit more than the previous node's ID did, starting at the last digit (suffix-based routing). The management of the overlay network focuses on fault tolerance: soft stating, time-outs, and republishing to ensure accuracy of the information, triple redundancy and back-pointers in the routing tables, use of several "root" servers, i.e. redundancy in the nodes responsible for a key.
- *Pastry:* The basic concept and topology is the same as for Tapestry, except that prefix-based routing instead of suffix-based routing is applied. The fault-tolerance focus is replaced by an apparently more light-weight scheme.
- *Chord:* Chord arranges keys and nodes around an identifier circle. The node with the largest number preceding the search key is responsible for holding it. Nodes maintain overlay links to a couple of successors and fingers as chords in the circle in exponentially increasing distances from the respective node, enabling to halve the remaining ID search space at each routing step. This becomes very similar to Tapestry and Pastry when choosing a base of 2 in the latter ones.
- *CAN:* CAN is based on a d-dimensional Cartesian coordinate space (or d-torus) separated into bins of varying size to implement a distributed hash table. Other than Tapestry, Pastry, and Chord, the node degree is thus fixed.

HyperCuP [14], takes a different approach. Like in Gnutella, flooding is used for the lookup. However, the overlay network is actively managed as a hypercube with good symmetry, diameter, and bisection width properties. It seems to be possible to also use a hashing scheme to improve routing efficiency. Furthermore, the authors propose an ontology-based routing scheme for the same reason.

Table 1 compares these systems (including AGILE) with respect to major requirements from Section 2 according to the developers' information or information deduced from algorithm descriptions. For all systems N denotes the number of nodes in the system, b and d are design parameters, where b is the base value for a digit representation of hash keys (where used) and d is the dimensionality of the CAN torus.

All mechanisms except CAN achieve logarithmic scalability with respect to the path length of a routing request or the network diameter. Chord does not allow to trade off the node degree for a lower number of hops by choosing a base higher than 2. Particularly for PC nodes, a higher node degree can easily be accommodated while allowing to reduce bandwidth and latency. While the existing algorithms only have a statistically inherent pruning probability related to their base b, they all achieve a routing efficiency of 1 - HyperCuP with its flooding mechanism being the obvious exception. The node degree scales logarithmically except for CAN, where it even remains constant. However, this limits the flexibility when a network grows. As to the wire length, Pastry, Tapestry, and CAN introduce optimization schemes. The methods and simulations to obtain figures for the stretch (*i.e.*, the relative latency of overlay routing compared to IP routing) are too different to base a good comparison on them. Several further proposals have been made to address the issue of wire length separately [3], [17], [18], [11], and [10].

Table 1 compares fault tolerance in terms of two dimensions, key redundancy and link redundancy. While replication can be controlled by the application, the lookup algorithms propose different mechanisms to conveniently place k replicas. For increased fault tolerance with respect to routing, Pastry and Chord keep redundant state information for closest neighbors or successors in the ring, respectively, whereas CAN and Tapestry set up (3 or r, respectively) independent entire routing tables. Tapestry further increases fault tolerance through soft-stating and heart-beat protocols.

Maintenance complexity, which is the number of messages per node join or leave, scales logarithmically for all systems but CAN. More detailed quantitative information is not available, but it is obvious that Tapestry with its surrogate routing and routing table redundancy will exhibit a higher complexity than the other mechanisms. As all algorithms build a probabilistic but fairly symmetric topology heterogeneity is only partly addressed by Tapestry through the BROCADE extension [17], and by CAN through load-dependent bin splitting.

Table 1. Comparison of Lookup Mechanisms

Characteristic	Tapes-try	Pastry	Chord	CAN	Hyper-CuP	AGILE
Network diameter	$O(\log_b N)$	$O(\log_b N)$	$\approx \log_2 N$	$O(dN^{1/d})$	$O(\log_b N)$	$\approx \log_b N$
Pruning probabilit.	1/b	1/b	1/b=1/2	n/a	n/a	1/b+37%†
Routing efficienc.	1	1	1	1	0	1
Node degree	$O(b^* \log_b N)$	$O[(b-1)^* \log_b N]$	$O(\log_2 N)$	O(d)	$O(\log_b N)$	$O[(b-1)\log_b N]$
Wire length/stretch	(≈ 2-4)	(≈1.3-1.4)	n/a	(≈ 2-3)	n/a	(≈ 2-4‡)
Key/replica redundancy	k salt values	k closest nodes	k succ. nodes	k hash functions	n/a	k salt values‡
Link redundancy	tripl. table entries	r closest neighbors	r succ. nodes	r realities	n/a	triple table entries‡
Maintenance complexity	$O(\log_b N)$	$3b^*\log_b N$	$O(\log_2 N)$	$O(N^{1/d})$	$O(\log_b N)$	$O(\log_b N)$‡
Fairness measures	none	none	none	none	none	virtual nodes
Symmetry / heterogeneity	symm.	symm.	symm.	symm./ bin split	symm.	symm./ GoI

AGILE creates a topology where each node can be the root of a tree. It exhibits similar network diameter and node degree characteristics as Tapestry. It adopts the advantages of Tapestry in terms of fault tolerance and wire length as well as its overlay maintenance scheme. However, AGILE considerably improves the pruning probability by applying an adaptive algorithm that brings requestors and requested keys stochastically closer together. Furthermore, it introduces fairness into the lookup mechanism by imposing the highest routing burden on those peers making the most frequent requests.

† For large b, otherwise 37%/(1-1/b); for assumptions, cf. Section 4.5
‡ Tapestry mechanism adopted

4. THE AGILE ALGORITHM

The AGILE algorithm proposed has been derived from the requirements presented above and combines the advantages of a scalable, hashing-based algorithm and topology with the efficiency and fairness of an interest- and usage-based group topology. The basic algorithm of the lookup inseparably combines the overlay topology and the lookup request routing.

For the subsequent discussions, consider the following scenario, where a peer node (the requestor) tries to find a certain service or content in the P2P network. It has to specify what it is looking for and the P2P system should return the content or service or a link to the content or service, *e.g.*, the IP address of a peer where it can be found. The desired and returned object is termed a lookup key (or simply key) and the specified request a lookup identifier (ID). Peer nodes in the network are characterized by their node ID, the node holding the lookup key is called provider node. Routing is the process of finding a path from the requestor to the provider node (which is usually unknown to the requestor) in a distributed way by forwarding lookup requests from one peer to another. The overlay network defines the structure on which request routing can take place.

4.1 ID Space and Arrangement of Nodes and Keys

A proper assignment of IDs to nodes and keys can be derived from the routing efficiency requirement. In order to avoid any kind of flooding and achieve a routing efficiency of 1, the P2P system is required to have global knowledge on the translation of search request or lookup key into lookup ID and on the association of the lookup ID with the provider ID. The use of hash functions, *e.g.*, based on SHA-1 [5] or MD-5 [12], to translate the search request, *e.g.*, the file name, into the lookup ID solves the first problem. The second problem is solved by arranging peer nodes in the same identifier space as the lookup IDs, *e.g.*, by applying the same hash function to nodes' IP addresses. The node with an ID numerically closest to the lookup ID will be the provider peer.

Figure 1 illustrates the identifier space in AGILE with peer nodes and lookup keys arranged in the same space. Note that due to the pseudo-randomness of the hash function distances of peers and the number of keys associated to a provider can vary. Stochastically, however, their distribution will be uniform. Figure 1 also introduces a hierarchy of types and genres in the identifier space. This hierarchy is derived from the requirement to achieve a good pruning factor. Assuming that request routing takes place along the identifier space (which, even though not linearly, is the case for AGILE), a good pruning factor requires that providers (or lookup keys, respectively) and potential requestors be located close to each other. AGILE achieves this through a clustering of keys and nodes into Groups of Interest (GoIs).

For a detailed illustration, assume a segmentation of lookup keys (content or services) as described by the following meta-information:

– Type, *e.g.*, music files, news information, or storage services.

- Genre, *e.g.*, rock, pop, classic, or house.
- Name, *e.g.*, RollingStones_Satisfaction or Beethoven_9.

Note that the specifics of the segmentation are purely illustrative and not focus of this work. One could as well apply a two-level hierarchy only, or subdivide the music genre further into different styles as done at allmusic.com or iuma.com, or use even higher level genre hierarchies [8].

Peer nodes have to be arranged in the same segmentation as content keys; in the illustration: type, genre, name, or, for nodes, IP address. The type and genre of a peer refer to its pre-eminent interests (its GoI). Section 4.4 below discusses how to determine the GoI of a peer and how to handle multiple interests. Hashing is then applied to each of the hierarchy levels. The lookup ID becomes TypeID.GenreID.NameID while the node ID will be TypeID.GenreID.AddressID.

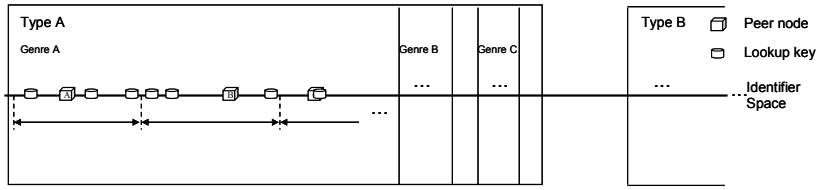

Figure 1. Identifier Space in AGILE

A total identifier space of 128 bit will be sufficient for most P2P systems. A distribution of bits to type, genre, and name/address, respectively, depends on the expected number of different types, different genres within a type and names/addresses within a type and genre. It is assumed that 32 bit each for type and genre and 64 bit for name/address will meet most demands.

4.2 Overlay Network Structure and Request Routing

Within the identifier space defined above, lookup requests have to be routed towards a node with the corresponding ID. It would be possible to route a request directly from one node to an adjacent one in the ID space in the direction of the lookup ID, who forwards it to its neighbor and so on until it finally reaches the provider. As this is highly inefficient and not scalable nor robust, an overlay network of virtual links needs to be constructed according to the requirements in Section 2, enabling every peer to route a request to any other peer in the identifier space with as few hops as possible.

A tree topology yields a good trade-off between node degree and network diameter. The tree is an efficient structure for searching or lookup, and both the node degree as well as the diameter scale logarithmically. For symmetry reasons and also to increase the bisection width of the graph, however, the simple tree structure needs to be extended: every peer has to be allowed to become the root of the tree or be on any other level, rather than maintaining links only to one level in the tree hierarchy.

Figure 2 (left) shows an AGILE overlay lookup tree. The lookup key segmentation defines the high-level tree hierarchy. As a root node, each peer

maintains links to peers from all different types. Within its own type, each peer maintains links to peers from all different genres. Within its own type and genre, each peer maintains links to all peers. This enables an efficient hierarchical lookup request routing from the more generic type to the more specific genre and, eventually, name.

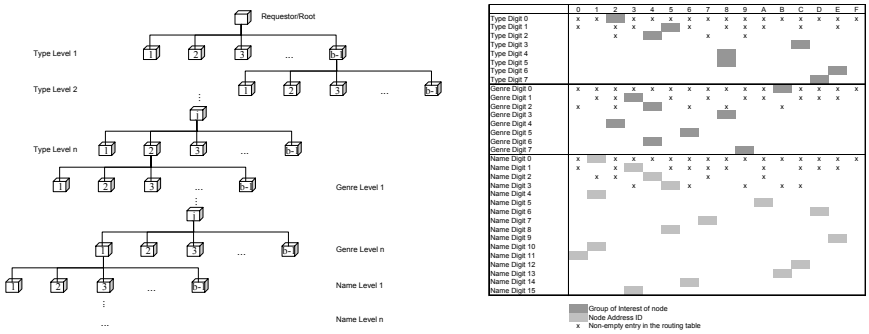

Figure 2. Left: An AGILE Overlay Lookup Tree; Right: Illustrative Routing Table

As the number of nodes in a genre or type can potentially become very large, a subordinate hierarchy is introduced to reduce the node degree, with a maximum of b nodes on each tree level. It is straightforward to associate b with the base of a numerical representation of the node or lookup ID. The position of a node (or key) in the tree is then determined by the succession of digits of its ID.

The resulting overlay network graph is defined through the virtual links on each peer, i.e. the routing tables. Figure 2 (right) illustrates a peer node routing table for a base b=16. The first row corresponds to the node being the root in a lookup tree. It has each one entry for peers with a different first digit in their ID. The second row holds entries for a lookup tree where the peer node is on the second level pointing to peers with identical first but different second digits. In general, the i-th row in the table points to peer nodes who have (i-1) digits in common with the peer in consideration and span the entire value space (b values) for the i-th digit, if all such nodes exist in the system.

Once the overlay topology is created, it is important to define how lookup requests can be routed from the requestor to the provider. This becomes very straightforward and efficient in the AGILE structure. Figure 3 illustrates the approach. At each hop, the routing peer forwards the request to a peer such as to match one more digit of the node ID, starting at the first digit. To simplify the illustration, Figure 3 only represents the first three digits.

For example, consider a peer requesting a key with an example ID 12345678.12345678.1234567890ABCDEF. The requesting peer looks into the first row of its routing table for a peer with “1” as a first digit and sends the request. The contacted peer looks into the second row of its routing table and forwards the request to a peer with “2” in the second digit, while the routing entries in the second row automatically ensure that the first digit of all entries is “1”. The process continues until the type ID is matched or the search is stopped. The same

mechanism runs for the genre ID. Finally, for the name ID, the process stops, when it reaches a peer with an empty corresponding row in the routing table. This peer holds the key, if it exists, or returns an error message. It is obvious that a requestor directly starts with the search for the name ID, if it itself belongs to the corresponding GoI. Similarly, a request may progress several digits at a time if the lookup ID matches more than one further digit with the processing peer. The pseudo-code for AGILE routing can be found in [7].

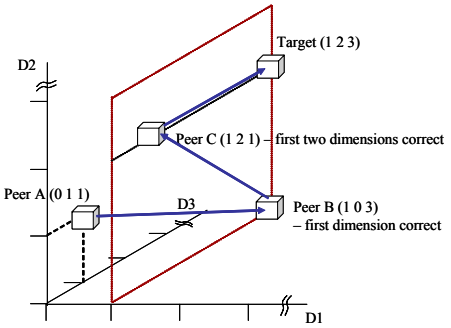

Figure 3. Illustration of Topology and Routing in AGILE

4.3 Insertion and Removal of Keys and Nodes

In order for the mechanisms described in the previous paragraph to work, it is necessary to first insert keys into the system and onto the node with the numerically closest ID. Furthermore, the topology (*i.e.*, the routing tables) have to be maintained as peers join and leave the network.

The insertion of keys into the system works exactly reciprocal to the lookup of a key. The peer node wishing to offer new content or services initiates an insert request with the according lookup ID. The request is routed just in the same way as a lookup request until it reaches the designated provider peer node which stores the key. For the removal of a key, the peer that stops to offer certain content or services sends a removal request with the according lookup ID into the network. The provider peer deletes the key.

The insertion of nodes into the system also works along the routing path. The new node contacts any known node. A node insert request is routed according to the usual routing procedure with the joining node's ID as lookup ID. At each hop in the path, the existing node learns about the new node. The joining node, in turn, can copy a row (row i at the i-th hop) from the forwarding node's routing table to initialize its own routing table. The insertion of nodes becomes more intricate once one wants to optimize wire length and achieve proximity in the underlying network for all or most nodes in the routing table. We have adopted the Tapestry [16] and Brocade [17] mechanisms including the algorithms for node removal, redundancy creation and fault management and the replication strategy.

4.4 Group Management and Adaptiveness

Groups of Interest (GoI) have been introduced to achieve a good pruning probability or ”tunneling”, since the first hops are avoided through GoIs. The goal of adaptive GoI management is to establish a process for peers joining and leaving GoIs such as to improve pruning or tunneling while keeping the overhead for group management itself reasonable.

A peer first joins a GoI by explicitly choosing categories of interest during the installation phase. Afterwards, requests for content will automatically make it join the requested GoI. That means, a peer can join more than one GoI. For each GoI, it carries a different node ID, derived from its GoI and IP address as discussed before. When joining a GoI and creating a new node ID, the peer effectively creates a new virtual node. It has to maintain a complete routing table for the virtual node that corresponds to its ID. The insertion takes place just as for a real node.

Two mechanisms help keep the overhead incurred by introducing virtual nodes and catering for more than one ID on a single node minimal: *thresholding* and *time filtering*. Thresholding means that a node only joins a new GoI, if the number of requests to that GoI exceeds a certain value. Time filtering means that the accounting of requests towards the threshold will be attenuated over time. Effectively, a node will leave a GoI, if it no longer makes requests to that group over a period of time - the corresponding virtual node is removed. Initially, time filtering will be a simple windowing; subsequent improvements are possible using adaptive filtering to predict future request behavior. It is assumed that the observation of a peer’s past behavior leads to reasonable predictions as the change rate of likes and dislikes will be slow compared to the request rate. In addition to thresholding and time filtering, a third mechanism, *aggregation*, may be required in designs choosing a higher-level hierarchy than the three level type-genre-name example, where requests to presumably very small leaf-GoIs occur only infrequently. Requests not only to leaf-GoIs in the hierarchy will be counted, but all requests within a higher- (up to second-) level hierarchy will be aggregated. Once the threshold for the aggregated requests is exceeded, a virtual node will be created at the leaf-GoI most requests have been made to.

Through the introduction of GoIs, their automated update, and the consequent introduction of virtual nodes, AGILE makes the lookup topology adaptive. Nodes eventually move toward the content they like and request.

The pseudo-randomness of the hash function in AGILE ensures load balancing, as nodes as well as content items are spread uniformly over the key space with respect to their type, genre, and name. However, GoIs in AGILE allow hot spots in the key space to form. If many nodes share a popular common interest, the key space will become far more populated in the respective type/genre area than in the areas corresponding to less popular interests. This, however, is a natural process. As the GoIs of these nodes coincide with their requests, the degree of node agglomeration is proportional to the degree of request agglomeration. Proper load balancing in the system is ensured.

Peers that have joined several GoIs, however, do have to carry a significantly higher routing load than others. This meets the system’s fairness requirement. Peers

requesting many content items from many GoIs and, thus, consuming many network resources also have an increased routing burden themselves. Peers making very infrequent requests to GoIs are not affected as the time filtering and thresholding makes them eventually leave the GoI in concern, releasing the additional routing burden.

Peers with frequent requests to the same GoI also carry a higher routing load in AGILE. New virtual nodes within the same GoI are automatically created when the number of requests per time interval exceeds a certain threshold. The pseudo-code for GoI-management can be found in [7].

4.5 Evaluation

A detailed evaluation of the node degree and the average number of hops for a lookup request is given here to show the impact of adaptive, group-of-interest-based overlay management on performance.

For the node degree, the routing table is considered. The routing table is densely populated in the first rows for type, genre, and name/node ID, depending on the number of nodes. As GoIs are spread uniformly across type ID and genre ID, respectively, it is unlikely that one GoI will have many identical digits with another GoI - the table becomes very sparse in the bottom rows. The same holds true for the name ID. More precisely, the probability that entry j in row i of the type, genre, or name area is populated is,

$$p_{i,j} = 1 - (1 - b^{-i})^{N_{t,g,n} - 1}$$

where $N_{t,g,n}$ denotes the number of different types, the number of different genres within a type, or the number of nodes within a GoI, respectively. Note that the counting of rows starts from 0 for each of the areas type, genre, and name. This yields for the total population of the table, the node degree d

$$d = (1 + n_v)(d_t + d_g + d_n); \quad d_{t,g,n} = (b-1) \sum_{i=1}^{R_{t,g,n}} \left[1 - (1 - b^{-i})^{N_{t,g,n}-1} \right]$$

where n_v is the number of virtual nodes and $R_{t,g,n}$ is the number of rows for type ID, genre ID, and name ID, respectively. The node degree is plotted in Figure 4 (left) for a base b=16, R_t=R_g=8, R_n=16, n_v=0. Two curves show the node degree for 50, and 5 different types and different genres within a type, respectively. Except for very low number of nodes, both curves lie well below the logarithmic curve $(b-1)\log_b N$ as well as below the reference curve without grouping (R_t=R_g=0, R_n=32), which can be regarded as an approximation for algorithms without grouping like Tapestry and Pastry.

The average number of hops for a lookup request n_h is approximated as follows:

$$n_h = (1 - p_{GoI,t})\, n_{h,t} + p_{success,t}(1 - p_{GoI,t,g})\, n_{h,g} + p_{success,t,g}\, n_{h,n} + p_{success,t,g} \cdot 1$$

where $n_{h,t}$, $n_{h,g}$, $n_{h,n}$ are the number of hops needed to match the type, genre, and name of the lookup key, respectively, if the lookup key exists, but does not fall within the requestor's group of interest. $p_{GoI,t}$ and $p_{GoI,t,g}$ denote the probabilities that the lookup refers to the requestor's group of interest type or genre, respectively.

$p_{success,t}$ and $p_{success,t,g}$ define the probabilities that the request is successful with respect to the type and genre.

The additional hop is an approximation for the hops that occur when the next digit cannot be matched, but when nevertheless closer nodes are available in the routing table. As type, genre, and address IDs are uniformly distributed, it is unlikely that more than one such hop occurs.

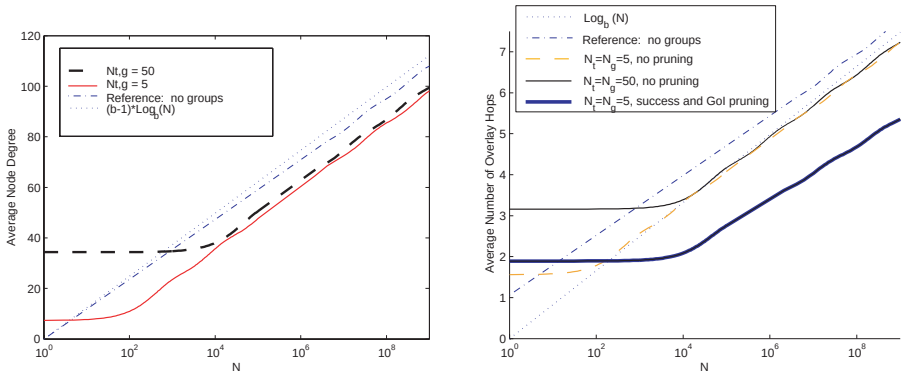

Figure 4. Left: Node degree; Right: Number of hops

Based on the likelihood that a node exists sharing i digits with the lookup ID, $p_{exist,i}=1-(1-b^{-i})^{Nt,g,n}$, it is:

$$n_{h,\,t,\,g,\,n} = \sum_{i=1}^{R_{t,\,g,\,n}} i(p_{exist,\,i} - p_{exist,\,i+1})TF_i$$

where $p_{exist,R+1}$ is defined to be zero. As some of the hops from one row to the next one happen on one and the same node and do not represent actual hops on the overlay network, the tunneling factor TF_i is introduced. It represents the ratio of hops on the overlay network to advances in routing table rows up to row i and can be derived to be

$$TF_i = \frac{b^{-i}}{i} \sum_{k=1}^{i} \binom{i}{k} k(b-1)^k = (1 - b^{-1})$$

The additional pruning factor achieved through the introduction of GoIs becomes

$$f_{p,\,GoI} = 1 - \frac{n_h}{n_{h,\,t} + n_{h,\,g} + n_{h,\,n} + 1}$$

The average number of hops is plotted in Figure 4 (right) for b=16, $R_t=R_g=8$, $R_n=16$ as for the node degree. As before, for comparison, $\log_b N$ and a reference curve without grouping ($R_t=R_g=0$, $R_n=32$), which should show similar results as algorithms like Tapestry or Pastry are also shown. Two curves for 50 and 5 different types and different genres within a type, respectively, show the expected number of

hops, when there is no pruning due to a node requesting a lookup key within its own GoI or due to quick abortion of the lookup when the type or genre is not available ($p_{success}$=100%, p_{GoI}=0%). For a reasonably large number of nodes, both curves are close to $\log_b N$ and exhibit a slight gain compared to the reference case. The last curve represents the average number of hops for 50 different types and genres within a type when there is pruning; the scenario assumes $p_{GoI,t}$=70%, $p_{GoI,t,g}$=30%, $p_{success,t}$=95%, and $p_{success,t,g}$=85%. In this case, the pruning factor becomes 37% compared to the reference case when there are 1 million nodes in the network.

Additional pruning gains can be achieved using a higher-level hierarchy for GoI structuring. Effects on the node degree will be limited and beneficial as long as no additional virtual nodes are created; this, however, is highly dependent on the thresholding and aggregation parameters applied.

The effects of an increase in amount and type of information in the system can be studied by comparing the graphs for $N_{t,g}$=5 and $N_{t,g}$=50 in Figure 4: the effect is limited and mostly visible in small networks only.

5. CONCLUSIONS AND FUTURE WORK

AGILE is a new lookup and routing algorithm bringing requestors and providers close together in Groups-of-Interest (GoI), tackling important scalability and performance concerns about the overlay network management for lookup and routing, while at the same time promoting system fairness. It can be applied to all peer-to-peer applications requiring such lookup services, as diverse as file sharing, distributed search and indexing, and, with some adaptations, distributed storage or file systems and distributed computing.

In future versions of the algorithm, searches with regular search expressions will be investigated. As a first step, search within a GoI can be replaced by controlled flooding rather than hash-based routing. Subsequently, a globally known semantic closeness operation will be needed to replace the hashing scheme, combined with proper load balancing, as the pseudo-random uniform load distribution due to hashing will be lost.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] H. Balakrishnan, M. Kaashoek, D. Karger, R. Morris, I. Stoica: *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*; ACM SIGCOMM, San Diego, August 27-31, 2001.

[2] E.A. Brewer: *Lessons from Giant-Scale Services*; IEEE Internet Computing Vol. 5 Nr. 4, pp. 46-55, July/August 2001.

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

[4] 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.

[5] FIPS 180-1, *Secure Hash Standard*; U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA, April 1995.

[6] K. Hwang: *Advanced Computer Architecture;* McGraw-Hill Series in Computer Science, p.77, 1993.

[7] J. Mischke, B. Stiller: *Peer-to-peer Overlay Network Management Through AGILE: Adaptive, Group-of-Interest Based Lookup Engine*; Extended Version, ETH Zürich, Switzerland, TIK-Report No. 149, August 2002.

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

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker: *A Scalable Content-Addressable Network*; ACM SIGCOMM '01, San Diego, 2001.

[10] S. Ratnasamy, M. Handley, R. Karp, S. Shenker: *Topologically-Aware Overlay Construction and Server Selection*; IEEE INFOCOM, New York, June 2002.

[11] S. Rhea, J. Kubiatowicz: *Probabilistic Location and Routing*; IEEE INFOCOM, New York, June 2002.

[12] R. Rivest: *The MD-5 Message Digest Algorithm*; RFC 1321, 1992, http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1321.html in August 2002.

[13] S. Saroiu, P. Gummadi, S. Gribble: *A Measurement Study of Peer-to-peer File Sharing Systems*; Technical Report # UW-CSE-01-06-02, Department of Computer Science & Engineering, University of Washington, Seattle, 2002.

[14] M. Schlosser, M. Sintek, S. Decker, W. Nejdl: *HyperCuP - Hypercubes, Ontologies and Efficient Search on P2P Networks*; International Workshop on Agents and Peer-to-Peer Computing (AP2PC), Bologna, Italy, July 2002.

[15] K. Sripanidkulchai, B. Maggs, H. Zhang: *Enabling Efficient Content Location and Retrieval in Peer-to-Peer Systems by Exploiting Locality in Interests*; ACM SIGCOMM, Computer Communication Review Vol. 30 Nr. 1, January 2002, p. 80.

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

[17] B. Zhao, Y. Duan, L. Huang, A. Joseph, J. Kubiatowicz: *Brocade: Landmark Routing on Overlay Networks*; First International Workshop on Peer-to-Peer Systems (IPTPS), Cambridge, MA, March 2002.

[18] 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.