Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Jan Mischke, Burkhard Stiller

Peer-to-peer Overlay Network Management through AGILE: An Adaptive, Group-of-Interest-based Lookup Engine

TIK-Report
Nr. 149, August 2002

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Jan Mischke, Burkhard Stiller:
Peer-to-peer Overlay Network Management Through AGILE: An Adaptive, Group-of-Interest-based Lookup Engine
August 2002
Version 1
TIK-Report Nr. 149

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

Peer-to-peer Overlay Network Management through AGILE: An Adaptive, Group-of-Interest-based Lookup Engine

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

Peer-to-peer (P2P) systems enable the direct communication between peers, which offer or request for services, resources, or content. While distributed peer-to-peer lookup services define the most important function in P2P systems, scalability and efficiency are key for those services. Currently, state of the art mechanisms actively create and manage a peer application layer overlay network to achieve this goal. The proposed mechanism AGILE (Adaptive, Group-of-Interest-based Lookup Engine) extends this management approach, where the overlay network adapts 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.

Keywords: *Peer-to-peer (P2P) Lookup Services, Overlay Network Management, Scalability*

1 Introduction

Peer-to-peer (P2P) networking, the direct cooperation of computing nodes at the edge of a network without a central server, has recently gained much attention again. While the Internet had originally been designed to the P2P paradigm, with Usenet as the most prominent P2P application example, the 90's have been dominated by the client/server paradigm and their applications.

By now, particularly the P2P file sharing services like Napster and Gnutella, but also numerous other P2P services, such as Seti@Home or Groove, are proliferating more rapidly than the fastest growing client/server services. The advantages of those P2P approaches are obvious: P2P enables the exploitation of idle resources at the edge of a network. This covers also their minimized administration rather than costly central server farms, and it can provide access to a vast content variety, supports user equality, and avoids censorship. However, apart from issues in P2P like quality control, security, or the lack of viable business models, the performance and scalability of P2P systems are a major concern. This applies particularly to the most distributed functionality in P2P systems: the lookup of content items or services within the P2P network.

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, while more sophisticated approaches, like Tapestry, Pastry, or Chord, actively manage the overlay network such as to ensure robustness and alleviate lookup and request routing. The latter systems, however, pay little attention to the heterogeneity of peers with respect to their interests and capabilities.

Concerning the internal structure of a peer-to-peer system, it builds on existing network infrastructure. For example, in the TCP/IP world, every node is connected to the global Internet. Similarly, in mobile networks (such as GSM or UMTS), it is possible to contact every mobile via the corresponding network layer protocol stacks. In theory, it would be possible to base a peer-to-peer platform entirely on the underlying network topology. Every network node would be assumed to be a peer, routing would be performed according to the standards of the network.

However, several reasons exist for choosing a better approach and introducing a logical overlay network on top of the underlying infrastructure. Firstly, almost all available networks build a strict hierarchy, contradicting the idea of peers being all equal. Secondly, not all nodes in the Internet will participate in a peer-to-peer network. Thus, it is straightforward to arrange a network consisting only of the participating peers. Thirdly, routing mechanisms and paths from one participating peer to another can be complex and are proprietary to the network. Hence, an abstraction can not only help to simplify the P2P network, but also alleviates the portability of the platform to different networks or even enables interoperability. Therefore, an overlay network is constructed through the P2P platform/application layer routing or link information on each peer. A peer node has a (directed) link to another node, or, a neighbor, in the overlay network, if it knows the corresponding network (or transport) layer address. This knowledge is sufficient to establish a transparent end-to-end connection. This link can be described by characteristics of the end-to-end connection, such as bandwidth and latency.

The management of those P2P overlay networks remains to be dealt with efficiently. In case of the most simple P2P systems, like Gnutella, they do not actively construct and manage their overlay topologies. Moreover, Gnutella peers learn about other peers at random through ping requests and pong responds while adding (or removing) links and nodes in an uncontrolled way [6]. Unfortunately, the orderless structure requires a non-scalable flooding mechanism for lookup and the lengths of paths and node degrees can become large. It is more powerful to actively design an ordered topology according to a set of requirements and apply a distributed overlay management algorithm for the careful insertion and removal of nodes and links.

Therefore, the proposed mechanism AGILE (Adaptive, Group-of-Interest-based Lookup Engine) takes this approach. 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.

While this introduction demonstrated the importance of P2P overlay network management and topology design, particularly for lookup services, the remainder of this paper is organized as follows. Essential requirements for such a topology 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. Finally, Section 5 summarizes the work, draws final conclusions, and discusses future work perspectives.

2 Requirements of P2P Overlay Management

Based on the definition of the most important set of functional and performance requirements of a P2P system, key topological requirements for overlay networks and their management are derived.

2.1 Functional and Performance 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 systems has to be *robust* to frequent node joins and leaves and link failures.

2.2 Topological Requirements

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 [7]. The functional and performance requirements 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. Consider a non-symmetric topology like the classic tree: It is obvious that the root of the tree has a far more central role than all leaves. At the same time, symmetry assists load balancing. While in non-symmetric networks hot spots with high traffic load (the root in the tree) may exist, the load will balance over available connections in a symmetric network. Examples of symmetric topologies include rings, buses, hypercubes, complete meshes, cube-connected circles, or k-ary n-cubes.
  While symmetry appears to be one of the most basic requirements for a peer-to-peer topology, measurements as stated in [5] 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. Thus, it can make sense to explicitly design asymmetric overlay networks, where some peers adopt a server-like role.
- **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 (cf. below).
- **Bisection Width (β):** The number of connections from one part of the overlay network to the other part define its bisection width. Assuming proper load balancing (which can be ensured through symmetry, at least partly), the maximum throughput of the network is proportional to the bisection width (and the average bandwidth of a connection). Even more importantly, there is a direct relation between bisection width and 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. While a node degree higher than one is desirable for improved fault tolerance of the network from the perspective of a single peer, 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 according impact on memory consumption and processing power.
- **Wire Length ($\bar{\tau}$):** The wire length is the average round trip delay of a connection, contributing to the latency in the system (cf. below). The wire length is closely related to the issue of mapping an overlay network properly onto a physical network: A low wire length in a peer-to-peer overlay network can be achieved by choosing neighbors that are also neighbors or at least physically and topologically close in the underlying network.

2.3 Latency and Bandwidth Requirements

Clearly, with respect to the overlay network management for lookup services it is important to have a look in detail at latency and bandwidth consumption for a lookup request in order to assess the significance of these characteristics.

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

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

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

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

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. It is obvious that the packet size should be kept as small as possible. The equation also suggests that the node degree be kept low. However, this applies only if the routing efficiency is smaller than 1. A lower node degree automatically entails a larger network diameter. Note that a higher node degree also increases in principle the bandwidth available as it augments the number of links from or to a node, however, these links are only virtual links in the overlay network that all have to be mapped onto one and the same physical access line of a node.

3 Related Work

Tapestry [18], Pastry [4], Chord [1], and CAN [11] determine those 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.
- *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 [16], the lookup algorithm proposed for Edutella [8], 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 approach proposes an ontology-based routing for the same reason.

Table 1 compares these systems according to their developers' information (including AGILE) with respect to major requirements from Section 2. For all systems denotes N the number of nodes in the system, b and d are design parameters.

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 an 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 [19], [20], [13], [12], and [8]. The issue of fault tolerance is particularly emphasized by Tapestry with advanced redundancy and soft stating mechanisms. However, no results are available for its maintenance complexity, i.e. the number of messages per node join or leave, which scales logarithmically for all other systems but CAN. As all

Table 1: Comparison of Lookup Mechanisms

Characteristic	Tapestry	Pastry	Chord	CAN	HyperCuP	AGILE
Network diameter	$O(\log_b N)$	$O(\log_b N)$	$\sim\log_2 N$	$O(dN^{1/d})$	$O(\log_b N)$	$\sim\log_b N$
Pruning probability	~1/b	~1/b	~1/b=1/2	n/a	n/a	~1/b+~37%**
Routing efficiency	1	1	1	1	0	1
Node degree	$\sim b*\log_b N$	$\sim(b-1)\log_b N$	$\sim\log_2 N$	d	$O(\log_b N)$	$\sim(b-1)\log_b N$
Wire length/stretch	(~2-4)	(~1.3-1.4)	n/a	(~2-3)	n/a	(~2-4*)
Fault tolerance	++	+	+	+	0	++*
Maintenance complexity	n/a	$3b*\log_b N$	$O(\log_2 N)$	$O(N^{1/d})$	$O(\log_b N)$	n/a*
Fairness	-	-	-	-	-	++
Heterogeneity | symmetry	0 | symm.	symmetric	symmetric	0 | symm.	symmetric	0 | symm.

* Tapestry mechanism adopted
** For large b, otherwise 37%/(1-1/b); for assumptions, cf. Section 4.5
++: Requirement met to a very high degree; 0: medium; --: low

algorithms build a probabilistic but fairly symmetric topology; heterogeneity is only partly addressed by Tapestry through the BROCADE extension [19], and by CAN through load-dependent bin splitting.

Currently, HyperCuP seems inappropriate as long as flooding is proposed as its lookup mechanism. Further extensions might be interesting. While CAN's flexibility is limited due to the fixed dimensionality and consequently non-logarithmic scalability in path length, the fixed base 2 limits Chords flexibility. Pastry and Tapestry are very similar, however Pastry's more lightweight algorithm is replaced by a seemingly more fault tolerant one in Tapestry. Test results would be needed for a proper comparison.

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, wire length, and maintenance of the overlay. 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.

4 AGILE - An Adaptive, Group-of-Interest-based Lookup Engine

The AGILE algorithm proposed has been derived from those 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 in this chapter, 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 short: 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.

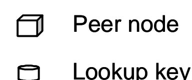
The discussion of AGILE is structured along the following major questions:

- How to arrange search keys and peer nodes in the identifier space (Section 4.1)?
- How should the overlay network look like to meet topological requirements and how are lookup requests routed on the overlay (Section 4.2)?
- How are lookup items and nodes inserted into the network or removed (Section 4.3)?
- How does the adaptive group management work and ensure load balancing, fairness, and heterogeneity (Section 4.4)?
- How does AGILE compare to its requirements and with related algorithms (Section 4.5)?

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 [14], 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, cf.



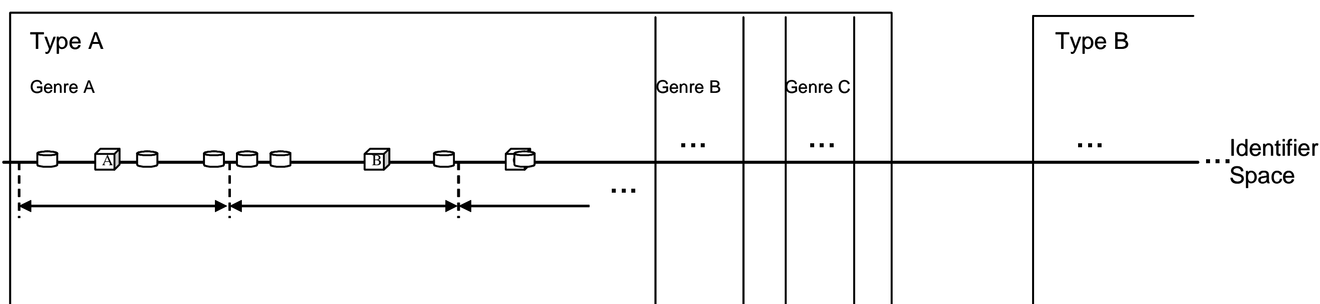


Figure 1: The AGILE Identifier Space

Section 4.2), 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.

RDF and XML [10] provide the appropriate functionality to improve this description and segmentation, however, this description task is not the focus of this work. Note, however, that peer nodes have to be arranged in the same segmentation:

- 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.

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 2.2, 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 (cf. [7] as well as Section 4.5). 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[1].

Figure 2 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.

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.

1. The topology finally being created is to some degree similar to a b-ary n-cube, i.e. a cube with n dimensions and b nodes in each dimension (see also Figure 4), where n is equal to the number of levels in the tree. In contrast to the b-ary n-cube, however, the b nodes in a dimension are fully connected. Furthermore, the cube is sparsely populated, many positions will be vacant as not all IDs will be assigned to nodes. Finally, in a b-ary n-cube, each two neighbors share all dimensions but one, whereas in AGILE, only the dimensions up to the one currently relevant for routing are common, the remaining ones are random. This is equivalent to having inclined links into the next dimension (rather than orthogonal ones).
This inclination has no effect on the routing performance compared to a b-ary n-cube topology as all relevant dimensions are kept correct. Bisection width will neither be affected as neighbors are chosen uniformly random through the use of hash functions.

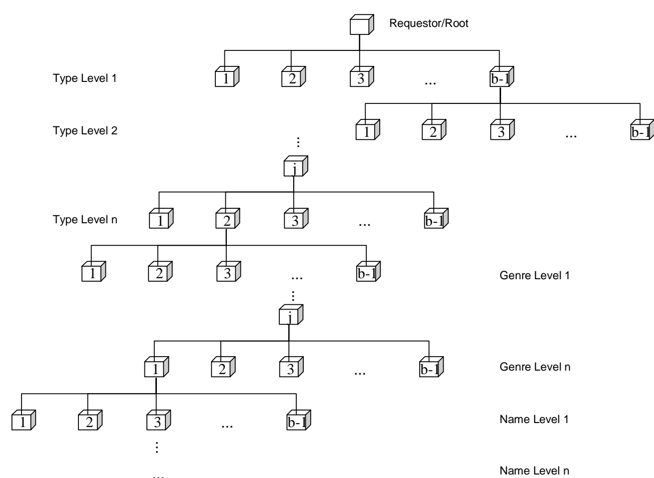


Figure 2: An AGILE Overlay Lookup Tree

The resulting overlay network graph[1] is defined through the virtual links on each peer, i.e. the routing tables. Figure 3 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.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Type Digit 0	x	x	GoI	x	x	x	x	x	x	x	x	x	x	x	x	x
Type Digit 1	x		x	x		GoI	x		x	x	x		x		x	
Type Digit 2			x		GoI			x		x						
Type Digit 3													GoI			
Type Digit 4									GoI							
Type Digit 5									GoI							
Type Digit 6															GoI	
Type Digit 7														GoI		
Genre Digit 0	x	x	x	x	x	x	x	x	x	x	x	GoI	x	x	x	x
Genre Digit 1		x	x	GoI		x		x		x	x		x	x	x	
Genre Digit 2	x		x		GoI		x		x			x				
Genre Digit 3									GoI							
Genre Digit 4			GoI													
Genre Digit 5							GoI									
Genre Digit 6					GoI											
Genre Digit 7										GoI						
Name Digit 0	x	ID	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Name Digit 1	x		x	ID		x	x	x	x		x		x	x	x	
Name Digit 2		x	x		ID			x			x					
Name Digit 3				x		ID	x			x		x	x			
Name Digit 4		ID														
Name Digit 5											ID					
Name Digit 6														ID		
Name Digit 7								ID								
Name Digit 8						ID										
Name Digit 9															ID	
Name Digit 10		ID														
Name Digit 11	ID															
Name Digit 12													ID			
Name Digit 13												ID				
Name Digit 14							ID									
Name Digit 15				ID												

(GoI) Group of Interest of node
(ID) Node Address ID
x Non-empty entry in the routing table

Figure 3: Illustrative AGILE Routing Table

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 4 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 4 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

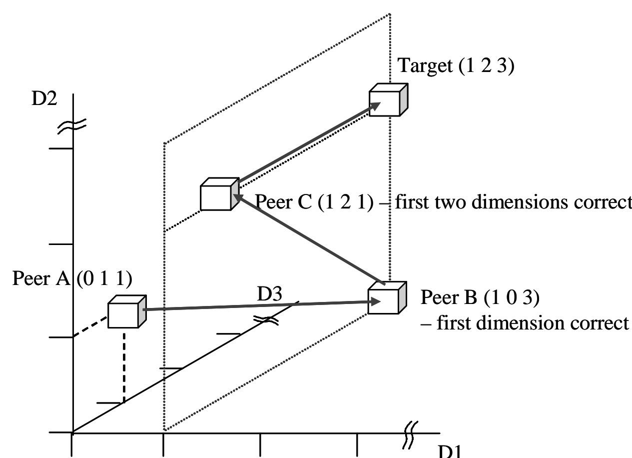


Figure 4: Illustration of Topology and Routing in AGILE

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.

Figure 5 shows the pseudo code for AGILE’s routing. The processing starts by checking whether the lookup key is already on the node and can be sent to the requestor. If not, type, genre, and name are matched one after another just as described above. If a better match for the lookup ID is available in the routing table, the request is forwarded to that peer, including a pointer to the row currently being processed. Otherwise an error message is returned to the requestor.

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

```
// lID: ID of lookup item
// node: Currently processing node
//  ForwardMsgTo(node, position) forwards the routing
request to specified node and hands over the current posi-
tion, i.e. row, being processed in the routing table

if key element KeysOnNode
    SendKeyTo(Requestor)
else{ // Forwarding necessary

if not lID.type == node.type { // Type
    while node[i] == lID[i]
          i++;
    if RoutingTable[i,lID[i]]
          ForwardMsgTo(RoutingTable[i,lID[i]], i)
    else
    SendErrMsgTo(Requestor, "NoSuchType");
}

else{ // Genre
if not lID.genre == node.genre {
    while node[i] == lID[i]
          i++;
    if RoutingTable[i,lID[i]]
    ForwardMsgTo(RoutingTable[i,lID[i]], i)
    else
    SendErrMsgTo(Requestor, "NoSuchGenre");
}

else{ // Name
    while node[i] == lID[i]
          i++;
    if RoutingTable[i,lID[i]]
    ForwardMsgTo(RoutingTable[i,lID[i]], i)
    else { // determine closest successor
          min = node[i];
          for j=0 To b-1
          if RoutingTable[i,j]
                if |j-lID[i]| < |min-key[i]|
                      min = j;
    if not min==node[i]
          ForwardMsgTo(RoutingTable[i,min], i)
    else
          SendErrMsgTo(Requestor, "NoSuchKey");
}}
```

Figure 5: Pseudo-code for AGILE Routing

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. It is proposed to adopt the Tapestry [18] and Brocade [19] 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 (cf. Section 4.1). 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.

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 Groups of Interest 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.

Some nodes, however, do have to carry a significantly higher routing load than others. Those that have joined several GoIs. 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.

Figure 6 shows the pseudo-code for the GoI management in AGILE. Upon each lookup request, 4 options are possible: either (a) nothing happens to the GoI management or the number of requests to that GoI is high enough to, (b) join the GoI, (c) create a virtual node if already a member, or (d) the drop-out of an old request entry makes the node leave that GoI.

4.5 Evaluation

A detailed evaluation of the node degree and the average number of hops for a lookup request is undertaken at this stage to assess 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

```
Function GoI_Update (LookupRequest) { // Called at each
                                      // request
    Account RequestGoI, Time of request;
    TF = TimeFilter(RequestGoI);
    if not RequestGoI element CurrentGoIs
                      // Examine if GoI join
         if TF >= JoinThreshold Join(GoI);
    if RequestGoI element CurrentGoIs
                      // Examine if new virtual node
         if (TF mod #CurrentVirtualNodes(GoI)) >
                      JoinThreshold JoinVirtual(GoI);

    if (#entries > MAX_Entries) or // Examine GoI leave
                      (age(OldestRequest) > Max_Age) {
         Remove(OldestRequest);
         TF = TimeFilter(OldestRequestGoI);
         if TF < LeaveThreshold Leave(GoI);
    }
}
Function TimeFilter(RequestGoI) {
    Array FilterCoeffs; // The filter coefficients, with
                      // lower weights for older entries
    Return Convolution(FilterCoeffs,
                      AccountEntriesMatchingGoI);
}
Function Join(GoI) {
    NodeID = GoIType.GoIGenre.IPAddress; //each hashed
    Send(InsertNodeRequest, NodeID);
    SetupRoutingTable(NodeID);
}
Function JoinVirtual(GoI) {
    NodeID = GoIType.GoIGenre.(IPAddress + #CurrentVirtu-
alNodes(GoI) + 1); //each hashed
    #CurrentVirtualNodes(GoI)++;
    Send(InsertNodeRequest, NodeID);
    SetupRoutingTable(NodeID);
}
Function Leave(GoI) {
    CurrentVirtualNodes(GoI)--;
    Send(RemoveNodeInform, NodeID);
    DeleteRoutingTable(NodeID);
}
```

Figure 6: Pseudo-code for AGILE GoI Management

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:

$$d = (1 + n_v) \cdot (d_t + d_g + d_n);$$

$$d_{t,g,n} = (b-1) \sum_{i=1}^{R_{t,g,n}} [1 - (1 - b^{-i})^{N_{t,g,n} - 1}]$$

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 7 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.

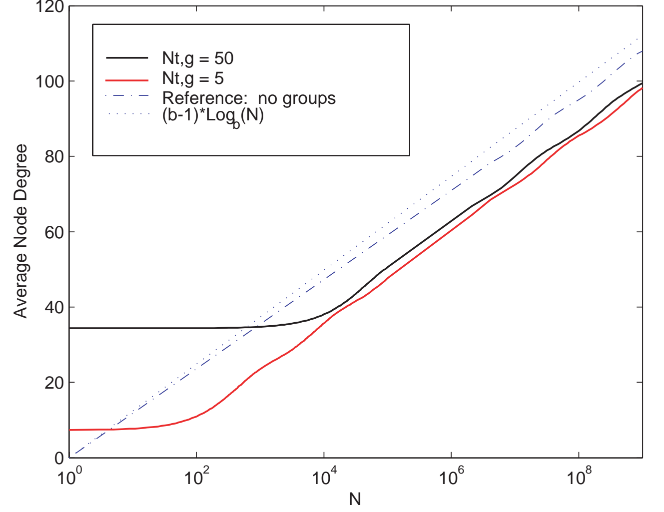


Figure 7: Node Degree

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

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

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.

Based on the likelihood that a node exists among all peers in the system that shares i digits with the lookup ID, $p_{exist,i} = 1 - (1 - b^{-i})^{N_{t,g,n}}$, 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 8 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.

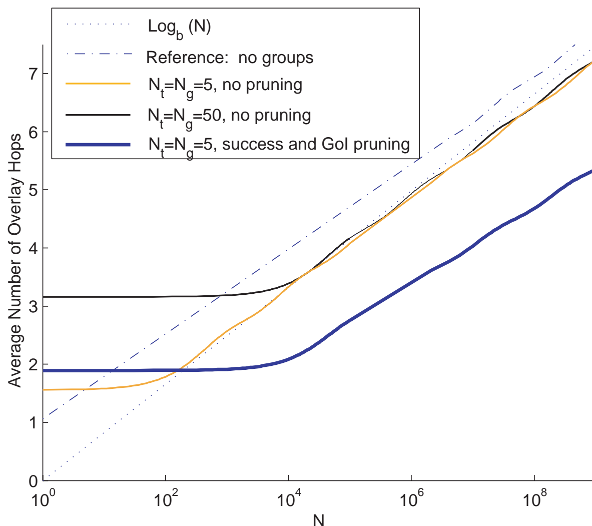


Figure 8: Number of Hops

In addition to the performance gains discussed above, AGILE's overlay management leads to good system fairness: The routing load imposed on a peer is made directly dependent on its resource usage in terms of lookup requests.

5 Summary, Conclusions, and Future Work

AGILE is a new lookup and routing algorithm with good performance characteristics. Lookup keys and nodes are arranged in the same ID space which is segmented according to content meta information and Groups of Interest (GoI), bringing requestors and providers close together. To meet topological requirements, AGILE creates and manages an overlay with some similarity to a b-ary n-cube, where each node can be the root of a tree. Subsequently, lookup or insert request routing comes down to a simple distributed tree search. AGILE adopts the advanced node insertion and removal schemes from Tapestry. GoI management based on requests made warrants proper fairness and load balancing characteristics and allows for the heterogeneity of peers.

As to the overlay network requirements of reasonable node degree and low diameter, they have directly been built into the topology design and show the desired logarithmic scalability. A good level of symmetry and a routing efficiency of 1 is ensured. Wire length and robustness characteristics are identical to Tapestry as the corresponding Tapestry algorithms should be adopted. The introduction of adaptive GoI management supports fairness and an additional pruning probability as high as 37% based on the assumptions in Section 4.5, while maintaining proper load balancing.

AGILE tackles important scalability and performance concerns about the overlay network management for the probably most distributed peer-to-peer functionality, lookup and routing. It has specifically been developed to allow efficient lookup of content or arbitrary services in the context of the MMAPPS "Market Management of Peer-to-Peer Services" architecture [8]. It can as well be applied to all other peer-to-peer applications requiring such lookup services, as diverse as file sharing, distributed search and indexing, and, with some adaptions, distributed storage or file systems and distributed computing.

Future work will tackle two issues. Firstly, the proposed and theoretically evaluated algorithm will be implemented, tested, and optimized. Secondly, AGILE currently requires the searching peer to exactly know the type, genre, and name of a content item or service. In future versions of the algorithm it will be investigated on how to allow searches with regular search expressions and enable the system to return lookup keys based on a best-match search rather than an exact lookup. In a simplistic approach, searches for all keys within a GoI and for names across all GoIs will be allowed. 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. Special thanks go to Jan Gerke and David Hausheer for joint discussions of the overall MMAPPS P2P architecture. Additionally, the authors like to acknowledge discussions with all of their project partners.

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, July/August 2001, pp. 46-55.

[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, pp. 329-350.

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

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

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

[8] *MMAPPS, Annex 1 - Description of Work*; Information Societies Technology (IST) Program, EU Fifth Framework Project, Project Number: IST-2001-34201, 2002.

[9] W. Nejdl: *Semantic Web and Peer-to-Peer Technologies for Distributed Learning Repositories;* Live Web Broadcast, June 17, 2002.

[10] A. Oram (edt.): *Peer-To-Peer: Harnessing the Power of Disruptive Technologies*; O'Reilly&Associates, Sebastopol, 2001.

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

[12] S. Ratnasamy, M. Handley, R. Karp, S. Shenker: *Topologically-Aware Overlay Construction and Server Selection*; 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), New York, June 2002

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.