Structuring Topologically-Aware Overlay Networks using Domain Names

Demetrios Zeinalipour-Yazti and Vana Kalogeraki
University of California - Riverside, Riverside CA 92521
{csyiazti,vana}@cs.ucr.edu

Abstract— **Overlay networks are application layer systems which facilitate users in performing distributed functions such as searches over the contents of other users. An important problem in such networks is that the connections among peers are arbitrary, leading in that way to a topology structure which does not match the underlying physical topology. This** *topology mismatch* **leads to large user experienced delays, degraded performance and excessive resource consumption in Wide Area Networks. In this work we propose and evaluate the** *Distributed Domain Name Order (DDNO)* **technique which makes unstructured overlay networks topologically-aware. In** *DDNO***, a node devotes half of its connections to nodes that share the same domain-name and the remaining half connections to random nodes. The former connections achieve good performance, because the bulk of the overlay traffic is kept within the same domain, while the latter connections ensure that the topology structure remains connected. Discovery of nodes in the same domain is achieved through on-demand** *lookup* **messages which are guided by a set of** *ZoneCaches***. Our technique is entirely decenntralized making it appropriate for use in Wide Area Networks. Our simulation results, which are based on a real dataset of Internet latencies, indicate that DDNO outperforms other proposed techniques and that it optimizes many desirable properties such as end-to-end delays, connectivity and diameter.**

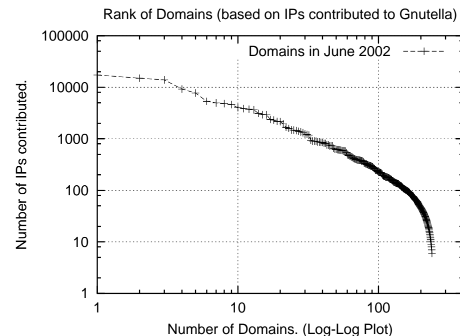

Fig. 1. Our analysis of the network traffic in [27] reveals that Large-Scale Overlay systems, such as Gnutella, consist of many thousands of nodes that belong to very few ISPs. More specifically we found that 45% of the nodes in a set of $244,000$ IPs belong to only 10 large ISPs and that 58% belong to only 20 ISPs.

I. Introduction

The advances of public networks in the last few years have increased the demand for Peer-to-Peer (P2P) application-layer protocols that can be used in the context of multicast [6], distributed object-location [20], [21], [22] and information retrieval [28]. Moreover, P2P file-sharing systems such as Napster [16] and Gnutella [8] have proven that large-scale distributed applications are feasible and that the P2P Computing model will play an important role in infrastructures of future Internet-scale systems.

P2P overlays can be divided into two categories: *Structured* and *Unstructured*. In *Structured* P2P overlays [20], [21], [22], network hosts and objects are structured in such a way that object location can be guaranteed within some hop count boundaries. In *Unstructured* P2P overlays on the other hand, hosts have neither global knowledge nor structure. Early unstructured systems, such as [8], rely on flooding the network with queries in order to locate the objects. Recently more efficient query routing techniques based on routing indices [7], heuristics [26] and caching [28] were proposed.

Overlay networks can also be used to provide services to the applications. For example, OverQoS [23] shows that it is possible to use overlay networks to provide QoS (such as statistical loss and bandwidth guarantees) without the need to modify the underlying IP network, focusing on streaming applications. RON [1] allows nodes to communicate with each other to detect path failures or perform path selection.

Therefore, an important problem is how to structure such an overlay network efficiently and in a decentralized manner where each node has only partial information. In the current topologies the connections between peers are not based on the underlying IP latencies, leading in that way to an inefficient overlay structure. This phenomenon leads to excessive resource consumption in Wide Area Networks as well as degraded user experience because

of the increased network delays between the peers in the overlay network.

In this work we propose and evaluate *DDNO (Distributed Domain Name Order)*, which is a distributed technique to make unstructured overlay networks topologically-aware. In *DDNO*, a node tries to connect to $d/2$ nodes that belong to the same domain (*sibling* connections) and to another $d/2$ of random nodes (*random* connections). The resulted DDNO topology achieves good performance through *sibling* connections while the additional *random* connections ensure that the topology structure remains connected. Discovery of *sibling* nodes is achieved through multicast *lookup* messages which are send out by each node and which traverse a set of *ZoneCaches* before finding other siblings. Our earlier study on the network traffic of the Gnutella [8] file-sharing network in [27], reveals that most of the participating nodes do belong to only a few ISPs (see figure 1). Therefore most nodes have a good probability of finding other *sibling* nodes which makes our scheme beneficial for the largest portion of the network.

Such an overlay can become the middleware component for a variety of network-based applications. In the context of *distributed file sharing* for instance, a user in Germany looking for traditional German music in the Bavarian dialect has a higher probability of finding some relevant answers if his search first spans in the German domains. If the overlay network is not topologically-aware, then the user's query will end up traversing domains across many different countries and continents, increasing therefore the delay of receiving back all answers and decreasing the probability of finding the desired results. Moreover, once the file is located the actual download time might also be very large as the file might physically reside far away from the user.

Furthermore, our scheme could increase the performance of *P2P Information Retrieval* [28] systems. In [28] we built and evaluated a large-scale decentralized newspaper network of 1000 nodes using 75 workstations. In this context, our topologically-aware scheme would enable users to span their queries to newspaper proxies that are closer to their locations enabling them therefore to locate local news. Although the necessity of topologically-aware overlays has been widely addressed in the context of structured overlays [4], [19], [25], [29], it hasn't received the same attention in the context of unstructured overlay networks.

Our Contribution

In this paper we consider a fully distributed technique for addressing the problem of efficient overlay construction in unstructured networks. More specifically:

- We propose and evaluate *DDNO (Distributed Domain Name Order)*, which is an efficient, scalable yet simple technique for constructing topologically-aware overlay topologies. DDNO is entirely distributed, requires only local knowledge and therefore scales well with the size of the network.
- We provide an extensive experimental study to evaluate the performance of our technique. In addition, we compare our technique with other heuristic-based techniques. Our results indicate that *DDNO* improves many desirable properties such as low end-to-end delays, connectivity and low diameter.

The remainder of the paper is organized as follows: In section II we describe a centralized version of the DDNO algorithm (DNO) and compare it with other heuristic-based algorithms for overlay construction in a centralized setting. In section III we present the DDNO technique and in section IV we present our experimental results. In section V we discuss related work and conclude the paper in section VI.

II. Centralized Overlay Construction Techniques

To simplify the discussion, in this section we describe a centralized version of our algorithm (DNO). We also compare this algorithm to three other currently proposed algorithms that are either *centralized* or require that each node has global information (needs to know its distance to all other nodes). In a centralized setting, these algorithms require information about all n hosts in the system as well as complete information on the "physical" distances between respective pairs (i.e. an nxn IP-latency adjacency matrix). In a real setting this might not be feasible, as transient user populations might not allow us at any point to gather complete knowledge. Evaluating these centralized algorithms however, provides us with a lower bound on the performance of the compared techniques. In section III and IV we will discuss decentralized versions of these algorithms.

A. Description of Algorithms

The following algorithms take as an input a vertex set $V = \{1, 2, ..., n\}$ and construct a "virtual" overlay topology $G = (V, E)$, where E set represents the overlay connections between the V vertices. The following popular algorithms have been used for constructing overlay networks:

1) **Random Algorithm (RAN)**: In this algorithm, each vertex v_i selects its d neighbors by randomly

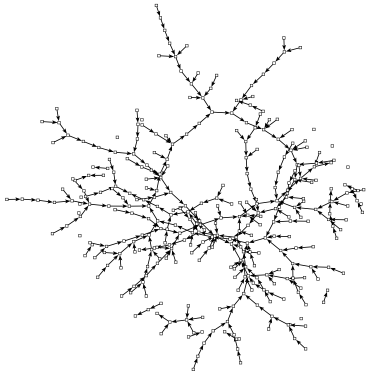

Fig. 2. Visualization of a Random graph with n=332 nodes (degree=2, diameter=32) using the Kamada-Kawai visualization model in Pajek [2]. Random topologies have the advantage that they are easy to construct and lead to connected topologies (if $degree > log_2 n$ [3]). The latencies at the overlay-layer however, usually don't match the underlying physical latencies.

choosing d other vertices. Since overlay connections are bi-directional it avoids connecting v_i to v_j if v_j is already connected to v_i. This is the algorithm deployed in most current P2P networks such as [8], [13] and its main advantages are that it is simple, does not actually require the nxn IP-latency matrix and leads to connected topologies if the degree $d > log_2 n$ [3] (see figure 2).

2) **Short-Long Algorithm (SL)**: The Short-Long algorithm, which was proposed in [19], alleviates the network unawareness of the RAN algorithm in the following way: Each vertex v_i, selects its d neighbors by picking the $d/2$ nodes in the system that have the shortest latency to itself (these connections are called *short links*) and then selects another $d/2$ vertices at random (these connections are called *long links*). Therefore SL requires the nxn IP-latency in order to find the latencies between the various node pairs. The intuition behind this algorithm is that the $d/2$ connections to "close-by" nodes will result in well-connected clusters of nearby nodes, while the random links serve to keep the different clusters interconnected and the overall graph connected.

3) **Short Algorithm (SS)**: In order to emphasize that by only selecting the shortest latency nodes might have a negative effect for the overall network structure; we also propose and study the Short (SS) algorithm. In SS, each vertex selects as its d neighbors the ones that have the shortest latency to itself (i.e. only short links). This, as we will see in II-D, will always result in disconnected topologies.

4) **Domain-Name Order Algorithm (DNO)**: In the Domain-Name Order Algorithm, which is the centralized version of the algorithm we propose in section III, a vertex v_i selects its d neighbors by picking the $d/2$ vertices that have the same domain-name with v_i. It then selects another $d/2$ neighbor at random. The idea of this algorithm is similar to the *SL* algorithm, in that we want to create well-connected clusters of nodes that are topologically close to each other without jeopardizing network connectivity. The main advantage however, is that the *DNO* algorithm does not require any global knowledge, such as the nxn IP-latency matrix, that SL needs.

B. Evaluation Parameters

Before evaluating the various generated overlay topologies G, we define four different evaluation parameters. First we define the Aggregate All-Pair Shortest Path (AggAPSP) parameter, which is the sum of all distances (pairs of shortest paths) on the overlay graph G. Formally *AggAPSP* is defined as follows:

$$AggAPSP_G = \sum_{i=0}^{n} \sum_{j=0, i \neq j}^{n} APSP[i][j], (APSP[i][j] \neq \infty) \tag{1}$$

where $APSP$ is an nxn matrix that stores all the minimum distances between all pairs. Such a table is obtained by running some All-Pair Shortest Path (APSP) algorithm[1] on the set of pairs in the edge set E. $AggAPSP$, can be thought as the end-to-end delay between all different pairs, and that is the reason it needs to be minimized. Although routing of messages on an overlay is performed based on the routing policies defined by each node, we use shortest path routing (similarly to [19]) which provides lower bounds for paths taken by packets.

Formula 1 however, does not take into regard the fact that some connection between overlay nodes might not be available. This happens in the case that the overlay network G is segmented into two or more partitions. Therefore we also define the $Clusters_G$ metric, which is the number of disconnected groups of nodes a given graph has. More formally:

$$Clusters_G = COUNT(Connected_Components) \tag{2}$$

[1] We use the Floyd-Warshall Algorithm [9].

where $Connected_Components$ is an algorithm that identifies the connected components of a graph[2]. It is important to mention that disconnected network segments are undesirable in overlay networks as this limits the reachability of nodes in the network.

The *Diameter* of an overlay G, which is the length of the longest shortest path between any two vertices v_i and v_j, is yet another parameter that needs to be taken account when evaluating an overlay topology. More formally $Diameter_G$ is defined in the following way:

$$Diameter_G = MAX(SP(v_i, v_j)), (\forall i, j \in V \text{ and } i \neq \text{j}) \tag{3}$$

where SP is the maximum shortest path between vertices v_i and v_j. Consider for example two overlay instances G_1 (ring topology) and G_2 (star topology) with the same number of vertices that have only different diameters δ_1 and δ_2 ($\delta_1 \gg \delta_2$). If an overlay message uses a parameter `TTL`, which limits the number of hops a message travels, then the nodes reached by the message are much less for G_1 than G_2. Therefore large diameters play a negative role in the resolution of some overlay message (e.g. some `QUERY` message) as those messages are required to travel more hops and possibly won't reach an adequate number of receivers.

C. Description of Datasets

Evaluating topologies based on the parameters outlined in the previous subsection requires a dataset in which the IP latencies are not synthetic. We therefore chose to base our experiments on the measurements of the *Active Measurement Project (AMP)* [10], at the *National Laboratory for Applied Network (NLAR)*. AMP deploys a number of monitors distributed along 130 sites to actively monitor the internet topology. AMP monitors ping and traceroute each other at regular intervals and report the results back to the project servers. Most of the current 130 monitors currently reside in the U.S with a few exceptions of some other International sites.

In our experiments we use an AMP 1.8 GB snapshot of traces obtained on the 30th of January 2003. The set includes data from 117 monitors out of which we extracted the 89 monitors which could be reversed DNS (i.e. given their IP we obtained a DNS name). We then construct the nxn IP-latency matrix (for all n=89 physical nodes), that contains the latency among all monitors. Since all 89 hosts are located at different domains, we choose to incorporate some degree of host replication per domain. Our study in [27] shows that hosts in

[2]We use the Component-Finding algorithm that uses DFS [9].

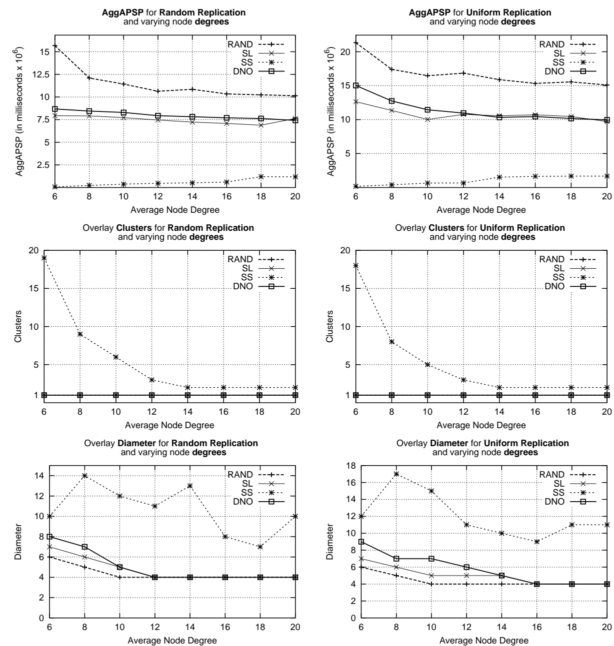

Fig. 3. Evaluating overlay topologies using data from the RR (left) and UR (right) datasets. By using DNO or SL we can significantly reduce the end-to-end delay between overlay nodes (top), while maintaining a connected topology (middle) with a relatively low *diameter* (bottom) for the same average degree.

a real overlay network, such as Gnutella, exhibit this characteristic. More specifically we choose the following host replication schemes:

1) **Random Replication (RR)**. We randomly replicate each host $[1..k]$ times. In our experiments we set $k = 7$ which consequently generated 332 nodes. This network attempts to address scenarios in which some domains contribute more hosts than other domains.

2) **Uniform Replication (UR)**. We replicate each host k times, for some parameter k. In our experiments we set $k = 4$, which consequently generated 356 nodes. This network attempts to address scenarios in which all domains contribute equally to the host distribution of the network.

D. Centralized Algorithms Evaluation

For the first experimental series we run the four centralized algorithms *RAN, SL, SS and DNO* using both the RR and UR datasets. The presented results are for different degree parameters larger than five (at which most algorithms stabilize to a single cluster). In Figure 3 (top row) we can see that all algorithms, other than SS, have a large *AggAPSP* value for smaller degrees but as the degree increases, AggAPSP quickly stabilizes. This happens because initially there are fewer paths

between the different node pairs. For example, if we have three nodes a, b, c connected in the following topology $a \overset{5}{\leftrightarrow} b \overset{10}{\leftrightarrow} c$ (where the number on the edges represents the latency between the respective nodes), then the addition of edge $a \overset{2}{\leftrightarrow} c$ will drop the *AggAPSP* from 30 (5+10+15) to 14 (5+7+2). SS on the other hand, presents always a very low $AggAPSP$ because: 1) each node only chooses the nodes that have the shortest latency to itself and 2) because the network topology is always disconnected and therefore many entries are not considered (i.e. APSP[i][j]=∞). The figure indicates that RAN has the highest *AggAPSP*, which means that it has the highest end-to-end delay between nodes. SL and DNO on the other hand are both able to perform much better because both algorithms choose half of their neighbors selectively, (i.e. the lowest IP latency and domain-name match respectively). On the same figure we can see that SL performs slightly better than the DNO algorithm but this is expected as SL has the advantage of choosing the $d/2$ "least latency" nodes while DNO has to rely on the domain-names as a metric for network distance. DNO however doesn't utilize the IP-latency table which provides the latencies between all pairs.

The fact that only RAN, SL and DNO generate connected topologies can be observed in Figure 3 (middle row). More specifically all three algorithms yield connected topologies while SS always results in disconnected topologies even for very large degree values (i.e. twenty). This happens because each node selects as its d neighbors only the nodes that have the shortest latency to itself.

Figure 3 (bottom row) shows the *diameter* of the four algorithms. As we can see only the SS algorithm generates topologies with arbitrary large diameters even in the case of very large degree parameters. On the same figure we can see that SL again slightly outperforms the DDNO algorithm but only for smaller degree parameters (less than 10). This is again expected as SL optimizes more the latency parameter (shortest path) between nodes, which consequently also decreases the diameter of the network. All experiments are averages of five executions.

III. DDNO - Distributed Domain Name Order Protocol

In section II we have seen that DNO might be useful for constructing connected overlays with low end-to-end delays and low diameters. The problem that arises in a real overlay setting is that we don't have global knowledge or not even a list of active users at all times. One solution would be to deploy some centralized lookup service, that, given some domain-name, returns IPs of other active nodes that belong to the same domain. Although such services are feasible, they are expensive, usually don't scale well and are vulnerable to denial of service attacks and censorship [16]. In this section we present the *Distributed DNO (DDNO)* algorithm which clusters nodes belonging to the same domain together without the need of centralized component that usually assists in the overlay construction process. An example of a DDNO Topology can be viewed in figure 4. Before describing the DDNO algorithm we will first describe two deployed functions: i) Split-Hash, which allows us to efficiently encode urls and ii) dnMatch which determines whether two domain names dn_1 and dn_2 belong to the same domain or not.

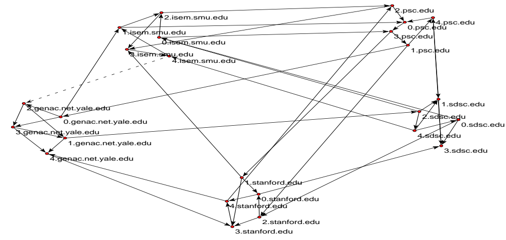

Fig. 4. A snapshot of a DDNO Topology with 25 nodes (degree=4) from 5 domains. Each node tries to connect to $\frac{d}{2}$ nodes in the same domain and another $\frac{d}{2}$ nodes in other random domains.

A. The Split-Hash and dnMatch Functions

Each node participating in a DDNO topology has some **Domain Name (dn)**, which is a string that conforms to the syntax rules of RFC 1035 [15]. Such a string, which is case insensitive, can be expressed with the regular expression $dn = label(.subdomain)^{+}$, where label and subdomain are some strings with certain restrictions, such as length and allowed characters. In order to determine whether two domain names dn_1 and dn_2 belong to the same domain we first introduce the *split-hash* function, which is a hashing function that splits a domain name dn into k hashes, where k is the number of subdomain strings in dn ($k = |subdomain(dn)|$). More formally *split-hash* is defined as following:

```
1: procedure SPLIT-HASH(dn)
2:     int size = |subdomain_dn|
3:     for j = 1 to size do
4:         result[j] = hash(m, subdomain_dn[j])
5:     end for
6:     return result
7: end procedure
```

where $hash(m, subdomain_{dn}[j])$, hashes the $subdomain_{dn}[j]$ using m bits. We chose to use hashcodes instead of raw domain-names because it allows us to keep the lookup message size small[3]. Furthermore, for performance reasons the hashcode doesn't need to be a non-colliding key[4] as this would again make ℓ prohibitively large. For example if we use a total of 160 bits for all the k generated hashes, then there would be an additional 100 bytes augmented to the lookup message after 5 hops. Instead, using a 20-bit hash function and assuming that keys are uniformly generated, we will be able to uniquely identify more that 1 million nodes and travel a distance of 40 hops with the same amount of bytes.

Now that we have introduced *split-hash*, we use the **dnMatch**(dn_1, dn_2) comparison function, which compares the individual *subdomains* hashes of dn_1 and dn_2. In the basic case, **dnMatch** returns $true$ if $dn_1 \neq dn_2$ and the subdomain of dn_1 and dn_2 matches. For example if dn_1="a.aol.com" and dn_2="b.aol.com" then dnMatch$(dn_1, dn_2) = true$. For dn_1="a.yahoo.de" and dn_2="a.yahoo.com" then dnMatch$(dn_1, dn_2) = false$. Of course our scheme can take advantage of the hierarchical structure of DNS and return the amount of similarity between two domain names (instead of using an exact match answer). For example if dn_1="a.rochester.rr.com" and dn_2="b.ny.rr.com" then dnMatch can return $\frac{2}{3} = 0.66$, rather than simply true or false.

The only limitation with *dnMatch* is that it can't distinguish two nodes that share the same *dn*, such as nodes in private networks using NAT (Network Address Translation). Although these nodes won't be able to connect to each other as siblings, they present only a small fraction of the nodes in networks such as Gnutella, in which they are less than 5% [27].

B. Joining a DDNO Topology

Let n denote a node which wants to join an overlay network N. Since n doesn't know which other nodes are currently active in N, it has to either probe nodes to which it was connected in some past session, or to consult some distributed discovery service D (i.e. some $hostcache$) which will provide n with an initial list of active nodes[5]. DDNO doesn't specify the details of the initial discovery part as its operation is application specific. It assumes that an out-of-band discovery service will provide n with a random list of active hosts $L = \{n_1, n_2, ..., n_k\}$, for some constant $k \geq \frac{degree}{2}$. It is important to notice that the individual hostcaches do not have global knowledge and therefore they can't be used for disseminating some pre-computed overlay structure or the distances between all node pairs.

Algorithm 1 Join_Network

```
1:  procedure JOIN_NETWORK(n, N)
2:      random ← 0
3:      while true do
4:          while (random < d/2) do
5:              if (empty(L)) then
6:                  L ← getRandomList(d/2)
7:              end if
8:              random ← connect(L)
9:          end while
10:         next ← getRandNode()
11:         L ← lookupDN(dn(n), ttl, next)
12:         wait(interval)
13:         connect(L)
14:     end while
15: end procedure
```

After n obtains the list L, it first attempts to establish a connection to $d/2$ random nodes, where d is the degree of n. The pseudocode of this procedure can be viewed in lines 4 to 9 of Algorithm 1. It is quite possible that some or all of the nodes n_i in L are not able to accept any new incoming connections. This might either happen because n_i reached its maximum degree or because n_i went offline. It this case n will need to obtain an additional list L from D. The next step is to find $d/2$ sibling connections (nodes which have a dnMatch with n). This is achieved by sending a `lookupDN` message to one of the existing (random) neighbors. The message will attempt to return a number of sibling nodes in N. We will discuss the complete operation of the `lookupDN` message in the next subsection. It is important to mention that a `lookupDN` message might get terminated without returning any results. Therefore a node might choose to pipeline several consecutive lookup messages.

C. Domain-Name Lookup in a DDNO Topology

We now focus our attention on the *lookupDN* procedure which is used by some node n, in order to discover other *sibling* nodes in N. We model the lookupDN message (denoted as ℓ) as a *multicast walker*. The goal of the multicast walker ℓ is to reach some node z that can guide it to the destination (i.e. a sibling of n). Note that before reaching z, ℓ may need to traverse a number of randomly selected neighbors. This can be viewed in figure 5, in

[3] RFC 1035 [15] defines that subdomain name must be 255 characters or less.

[4] Hash functions such as SHA-1 are 160-bit and collision of two keys is difficult.

[5] Both techniques are deployed in many Peer-to-Peer systems, such as Gnutella and Kazaa and work reasonably well. Hostcaches are either located on Web pages or dedicated servers.

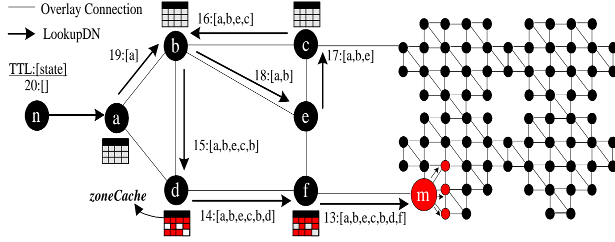

Fig. 5. **Domain-Name Lookup in a DDNO topology.** Each *lookupDN* message retains path information in order to populate the *ZoneCaches* of other nodes. The list appends shown on the lookupDN message in the picture illustrate the accumulated path in ℓ.

which ℓ takes the random itinerary $[a, b, e, c, b, d]$. At d however, ℓ is allowed to make an informed decision on which neighbor to follow next (in this example node f). This is achieved by using a special structure called *ZoneCache* that contains information on which nodes are reachable in a r-hop radius (it will be discussed in the next subsection). At the end of this procedure, ℓ is expected to reach some node m, which is a sibling of n. m then issues a *broadcast* message to all of its own *siblings*. Each of the receiving nodes, including m, will respond with a *LookupOK* message (denoted as ℓ') if they are willing to accept new connections. Therefore node n will end up receiving several answers out of which it will attempt to establish a connection to $d/2$ nodes, which will n's siblings.

One important problem with this approach is that ℓ might get locked in a cycle (e.g. loop $b \rightarrow e \rightarrow c$ in figure 5), limiting therefore its reachability. This might happen if the various nodes or the ℓ messages don't retain any state. An approach to overcome this problem is to keep some state at each node (e.g. a list with the identifiers[6] of the ℓ messages received recently). However, instead of retaining such state at each node, we choose to incorporate state information in ℓ as this also serves as an implicit mechanism to populate the ZoneCaches along ℓ's path. The state information included in ℓ, includes the **split-hash** h (described in section III-A), on the domain-name of each node that ℓ traversed (i.e. $state_\ell = \{h(v_n), ..., h(v_m)\}$).

The complete pseudocode of the $lookupDN$ procedure can be viewed in Algorithm 2. A node n sends a lookup message to node m using some ttl parameter, which determines the maximum number of hops that the given lookup should be forwarded. The ttl parameter, which is used in many networked applications, starts out from a predefined value and is decremented each time a lookup message is forwarded until it becomes zero.

[6] ℓ messages are uniquely identified by a random identifier that is locally generated by the issuer of ℓ.

Algorithm 2 lookupDN

1: **procedure** LOOKUPDN(n, ttl, m)
2:     **cacheRoute(n,ZoneCache(m))**
3:     **if** (dnMatch(n, m)) **then**
4:         **if** ((degree(m) $< d$) and not(connected(n, m))) **then**
5:             send(n,`"LOOKUPOK m.IP,m.PORT"`)
6:         **end if**
7:         broadcast(siblings(m))
8:     **else if** ($ttl > 0$) **then**
9:         **if** (hit(ZoneCache(m), hash(n))) **then**
10:             $next \leftarrow getNextNode(ZoneCache(m))$
11:         **else**
12:             $next \leftarrow getRandNode()$
13:         **end if**
14:         $lookupDN(n, ttl - 1, next)$
15:     **end if**
16: **end procedure**

D. The ZoneCache Structure

ZoneCache is a caching structure which is deployed locally at each node and its functionality is to guide `lookupDN` messages to their sibling nodes. A snapshot of such a structure is displayed in table I. The first column includes the hash of some domain-name and this information is extracted from parsing lookupDN messages. The second column indicates, the peer connection that will lead a future ℓ_2 to the corresponding destination, and the third column indicates the respective cost in hops. Finally ZoneCache also uses a timestamp parameter (fourth column) in order to limit the number of entries in the structure to a total size of C [7]. Once the repository of some node becomes full the node uses the Least Recently Used (LRU) policy to keep the most used entries in the cache.

The cache stores only the hashcodes of the nodes that are located within an r-hop radius in order to both limit its size and accuracy. We show how this works with the following example: Assume that node n sends a lookupDN message ℓ searching for some sibling and that this message reaches some node d (in figure 5). Also assume that ℓ has already passed from five nodes and that it has the following state: $state_\ell = \{a, b, e, c, b\}$. If the radius parameter of m's ZoneCache is set to three then node d will store the following quadruples (i.e. information for only three hops away): $\{(b, b, 1, ts), (e, b, 2, ts), (c, b, 3, ts), (a, b, 2, ts)\}$ where the first field is a hash of the destination node, the second field the next neighbor that leads to the destination, the third field the number of hops and the last field the timestamp parameter generated at the time

[7] We set ZoneCache's maximum entries parameter C to 350.

TABLE I
THE **ZoneCache Structure**. IT CACHES DOMAIN REACHABILITY INFORMATION FROM LOOKUPDN MESSAGES THAT TRAVERSE A GIVEN NODE.

Dest. Domain Hash	Neighboor	# Hops	TimeStamp
9A78DF	Socket3	3	10000000
421CDE	Socket1	2	10012000
...	...	...	...
2AB356	Socket1	2	10160000

of the record insertion. Please note that before storing the quadruples, we identify and eliminate cycles in the $state_\ell$ sequence (therefore $(a, n, 2, ts)$ is also considered). Furthermore if d's ZoneCache already contains any of the following hashcodes $\{a, e, c, b\}$ then ℓ would update some tuple only in the case that the new entry provides a shorter path to the respective entry. The next question is how the cached information becomes useful to some future lookup message. Suppose that node a sends a lookupDN message ℓ_2 to d (see figure 5) and that a and c are siblings (i.e. $dnMatch(a, c) = true$). Following the previous example, d has an entry in its zonecache which indicates that c can be reached through b in 3 hops. Therefore ℓ_2 will be routed towards c. Although neighboring ZoneCaches could actively exchange routing updates at regular intervals, like BGP, our passive caching scheme reduces significantly the amount of transmitted message and works well in dynamic environments as we will see in section IV.

E. DDNO Topology Maintenance

When a node disconnects from the DDNO topology it does not need to send any priori notification to the other nodes. This happens because each node continuously tries to maintain its degree to the pre-determined value d. If some *random* neighbor of n leaves N then n will either attempt to re-establish the dropped connection or find another node from the discovery service outlined in subsection III-B. On the other hand, if some *sibling* of n disconnects then n consults its *ZoneCache* in order to send the new lookupDN message towards a current sibling. It is expected that n will discover another sibling in 2 hops (as a node already maintains $(\frac{d}{2} - 1)$ siblings), which therefore makes the recovery of broken connections cheap.

Another technique would be to proactively exchange `lookupDN` messages with sibling nodes. Although such a strategy might allow a node to instantly react in the event of failures, it might become a large overhead for the overlay topology. For example our study in [27], on a collection of 56 million overlay messages obtained from the Gnutella network, reveals that 23% of all messages are `Ping` messages and 40% of them are `Pong` messages. `Ping/Pong` messages are the main technique for proactively discovering new nodes in the Gnutella Network.

F. Query Routing in a DDNO Topology

One of the major objectives of overlay networks is to facilitate users in performing distributed functions, such as queries over the contents of other users. In [28], we have made an extensive study on a number of different query techniques that can be applied in randomly generated topologies. In this work we propose the deployment of the DDNO topology which leads to more desirable overlay properties. Given that we have a DDNO topology some node might deploy any of the following search techniques: *Breadth-First-Search (BFS)* [8] (query all neighbors), *Random BFS* [28] (query a random subset of neighbors), *ISM* [28] (intelligently query a subset of neighbors) or *>RES* [26] (query the neighbors that returned the most results in the past). Our study which was performed a real network of 1000 nodes deployed on a network of 75 PCs reveals that by using our ISM technique we might be able to retain high degrees of recall while using only half messages and time used by the brute-force BFS technique.

DDNO allows multiple search algorithms to be deployed on top of its topology. The advantage of using DDNO is that the bulk of the incurred overlay traffic will remain within the same domain since only $d/2$ of the traffic will make its way to a different domain. Finally, the DDNO topology gives space for more sophisticated search techniques. In the context of a large-scale file-sharing application with many thousands of nodes, we might decide to forward query requests to only *sibling nodes*.

G. DDNO in a Hybrid Overlay Environment

Although the proposed DDNO topology leads to a flat topology, the basic approach can be utilized in some hybrid P2P environment such as Kazaa[13] and Gnutella[8] v0.6. In such an environment some nodes with long-time network connectivity and high bandwidth connections, known as *SuperPeers or UltraPeers*, form a backbone infrastructure which can be utilized by other less powerful nodes (denoted as *RegularPeers*). Such a model allows the network size to grow to millions of users because it differentiates short-time connection and modem users from other more powerful users (e.g. ADSL, cable modem users).

DDNO could be deployed in a hybrid P2P environment in the following way: A superpeer s initiates a lookupDN ℓ message to find $d/2$ other sibling and $d/2$ random superpeers. RegularPeers will again utilize the lookupDN message to discover the superpeer nodes that belong to their domain and that might be able to serve them. Of course using such an approach in an overlay, requires a large number of participating nodes, as smaller numbers would limit the number of superpeers the ℓ message locates. Therefore in the next section we present an experimental evaluation of the basic "flat" topology approach, rather than the "hybrid" topology discussed in this subsection.

IV. Experimental Evaluation

In this section we present our simulation-based environment and results of our evaluation. More specifically, we develop distributed versions of the centralized algorithms RAN, SL and DNO presented in section II-A. We deliberately don't present SS because it generates disconnected topologies. In a distributed setting some node has no topological information other than which are its own neighbors. Therefore global lists of other active nodes or IP-latencies are not available.

Since there is no fine-grained model of time in a simulation environment, we choose to divide time into units of algorithm *iterations*. During an *iteration* each node n is given the opportunity to establish connections to up to d neighbors. n is not assured that it might be able to connect to d neighbors in a single iteration. This happens because some or all of its attempts target nodes that have already reached their expected degree and therefore don't accept any new incoming connections. Therefore an algorithm may require several iterations before it stabilizes.

A. Description of Algorithms

Below we provide a brief description of the algorithms compared. They are named according to their centralized counterparts:

1) **Distributed Random Algorithm (DRAN)**: This algorithm is very similar with RAN in that each node n selects its d neighbors randomly but $DRAN$ uses the notion of *iterations* instead of allocating all nodes at once.

2) **Binning SL Algorithm (BinSL)**: BinSL is a distributed version of the SL algorithm proposed in [19] and discussed in section II-A. It again selects $d/2$ nodes at random and another $d/2$ nodes that have the shortest latency to itself. Since the adjacency-matrix of IP latencies is not available, *BinSL* deploys the notion of *distributed binning* in order to approximate these latencies. More specifically each node calculates the round-trip-time (RTT) from itself and k well-known *landmarks* $\{l_1 l_2 .. l_k\}$ on the Internet. The numeric ordering of the latencies represents the "bin" the node belongs to. Latencies are then further classified into $level$ ranges. For instance if the latencies are divided into 3 levels then; *level 0* accounts for latencies in the range [0,100), *level 1* for the range [100,200) and *level 2* for all other latencies. The level vector is then augmented to the landmark ordering of a node yielding a string of the type "$l_2 l_3 l_1 : 012$". It is expected that nodes belonging to the same bin will be topologically close to each other although *false positives* are possible, that is, some nodes do belong to the same "bin" although they are not topologically close to each other. The rate of *false positives* is a function of how many landmarks are used, as fewer will degrade the performance of the binning scheme. We therefore experimented with the following two datasets: i) **BinSL-4** which uses 4 landmarks with 3 levels and ii) **BinSL-12** which uses 12 landmarks and 3 levels.

3) **Distributed DNO Algorithm (DDNO)**: This is the technique we advocated in section III. We have experimented with various parameters for ZoneCache's *caching radius* and *ttl* parameters for lookup messages but present only the following most representative results: i) **DDNO-3** which uses lookupDN messages with a ttl of 20 and caching radius of 3. ii) **DDNO-5** which uses again a ttl of 20 but caches in a larger radius of 5.

B. Performance Evaluation

In the first experiment we evaluate the distributed algorithms against the three parameters we defined in section II-B (i.e. AggAPSP, Clusters and Diameter) using the *UR* dataset in which each node has a degree of 8. We obtained similar results for the *RR* dataset and therefore omitted their presentation. As we can see in figure 6 (left) DRAN has again the highest end-to-end delay as the AggAPSP stabilizes at 19M ms while all other algorithms perform much better. DDNO-3 and DDNO-5 use $\approx 13.5-14.5$M ms while BinSL-4 and BinSL-12 use $\approx 16-17$M ms. This means that DDNO-5 presents a 30% improvement upon the DRAN technique. We can also see that although we increase by three times the number of landmarks in the BinSL algorithm the accuracy of the binning scheme only increases about $0.8M$ ms.

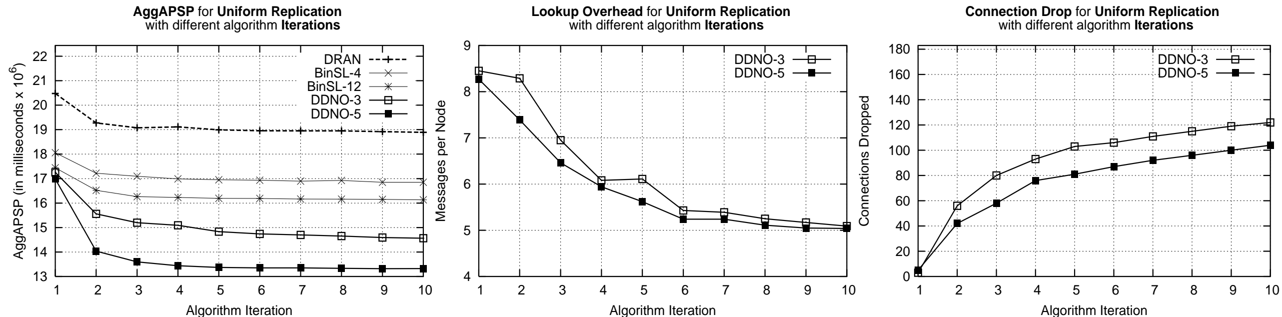

Fig. 6. **a) Performance Evaluation (left)** of DRAN, BinSL-4, BINSL-12, DDNO-3 and DDNO-5 using the UR dataset and **b) Overhead Evaluation (middle,right)** of the DDNO-3 and DDNO-5 techniques.

We can also observe that DRAN and BinSL manage to stabilize within the first few iterations as their operation doesn't involve temporary connections. We already discussed that DDNO maintains more that $d/2$ random connections if it is not able to locate $d/2$ siblings. Although this increases connectivity and prevents network fragmentation, it also slightly delays the stabilization process. We also experimented without allocating temporary connections and found that such an approach is viable, as it stabilizes after the seventh iteration, but it initially results in a very high AggAPSP. In this experimental series, and for the subsequent sections, all algorithms always generate connected topologies which therefore make the $Clusters_G$ evaluation parameter equal to one. Furthermore the $Diameter_G$ remains constant at five. Therefore the graph for both evaluation parameters is omitted. In the remaining subsections our goal is to investigate the overhead of DDNO technique, how it performs in a dynamic environment and how it scales with larger network sizes.

C. Overhead Evaluation

In order to assess the overhead of the DDNO technique, we investigate the average number of hops each lookupDN message ℓ traverses before finding some sibling node. These results are, as with the previous subsection, from the execution using the UR dataset. As we can see in figure 6 (middle), ℓ initally requires about eight messages (hops), before it is able to locate its siblings. In the subsequent iterations the various ZoneCaches get populated, which consequently lead more ℓ messages to the right regions. The plot indicates that after the sixth iteration, ℓ requires only five hops for both DDNO-3 and DDNO-5 although DDNO-5 stabilizes slightly faster because of the expanded coverage it offers.

The second overhead parameter that we investigate is the total number of temporary connections that are swapped with sibling connections once the latter are found. Figure 6 (right) indicates that DDNO-5 is again able to perform slightly better because ℓ messages are resolved faster, which consequently eliminates the need for temporary connections. We can further see that the total number of swapped connections for DDNO-5 and DDNO-3 is 100 and 120 respectively. This accounts to only a drop of $\approx$7% of the total connections in the case of DDNO-5 and $\approx 8.5\%$ in the case of DDNO-3.

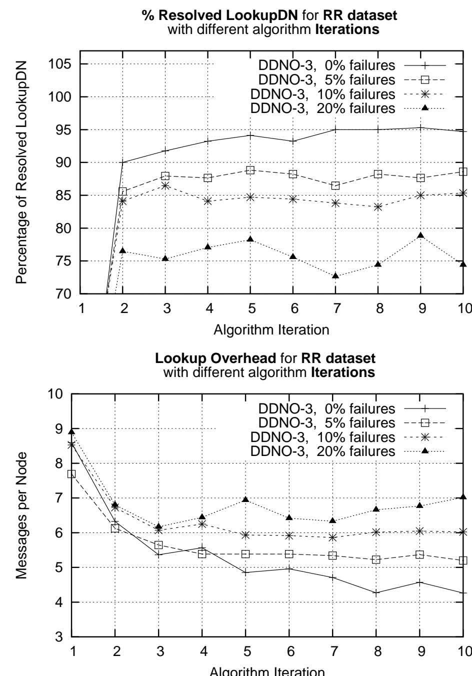

Fig. 7. Evaluation of the DDNO-3 algorithm over the RR dataset, in a dynamic network topology where nodes leave and arrive.

D. Dynamic Environment Evaluation

Network failures in overlay systems are commonplace because of the misusage exhibited at the application layer (e.g. users shut down their PCs without disconnecting) and the overwhelming amount of generated network traffic. Such failures generate a dynamic environment in

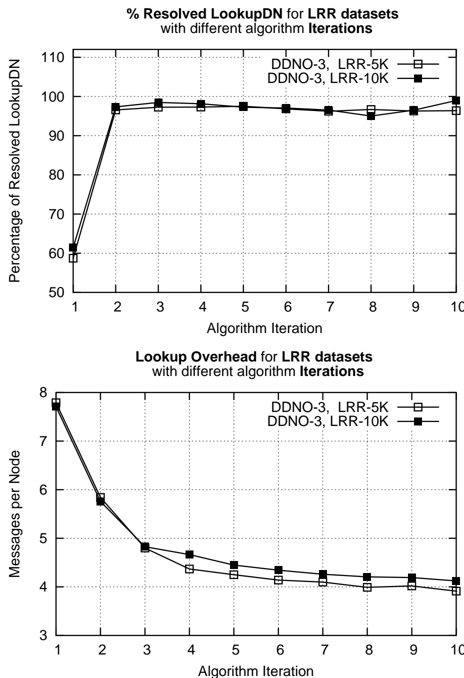

Fig. 8. Evaluation of the DDNO-3 and DDNO-5 algorithms over the LRR-5K and LRR-10K datasets which are networks of 5,000 and 10,000 nodes respectively.

which peers are leaving or joining the network in an ad-hoc manner. A highly dynamic environment neutralizes the purpose of the ZoneCaches, as cached information might become outdated before it gets the chance to be utilized.

We choose to evaluate only *DDNO-3*, where each node uses a ZoneCache with a 3-hop radius, since our preliminary runs on networks of different sizes, indicated that such a setting consistently offered good performance at a low overhead. In order to simulate network failures, we disconnect at each *iteration* a fraction of nodes. The failure rates we used are $\{0\%, 5\%, 10\%, 20\%\}$.

In figure 7 (top) we plot the number of resolved lookupDN messages after running DDNO-3 using the RR dataset. The figure indicates that ≈89% and ≈85% of the messages are resolved at 5% and 10% of failures respectively. Therefore low degrees of node failures don't significantly affect the performance of our scheme. With 20% failures the number of resolved lookup messages drops to 75%. Although this might be acceptable in some settings, the fact that the number of hops required by each message increases over time (see figure 7 (bottom)), might make our scheme not appropriate in such a dynamic setting. On the same figure 7 (bottom), we can also see that with 5% and 10% of failures the number of hops required by each messages stabilizes at 5 and 6 hops respectively. It is important to remind that in DDNO there is no explicit mechanism to delete outdated entries in the distributed ZoneCaches as this would introduce an additional messaging cost. Each node therefore relies on its LRU policy to invalidate old entries.

E. Scalability Evaluation

In this subsection we show how our technique scales to larger network sizes by measuring the percentage of resolved lookupDN messages and the average number of hops each message travels. For this experimental series we derive a new dataset from the Active Measurement Project data presented in II-C. More specifically we chose the Random Replication scheme and generate the **Large Random Replication (LRR)** dataset with 5,000 (**LRR-5K**) and 10,000 (**LRR-10K**) nodes. We used *DDNO-3* with ttl_ℓ=20 and nodes having a degree of 12.

In figure 8 (top) we can see that in the first iteration approximately 57% and 60% of the lookupDN messages are resolved for the LRR-5K and LRR-10K datasets respectively. This low rate is attributed to the fact that the various ZoneCaches are not populated adequately. In the subsection iterations however, the lookupDN procedure is able to resolve ≈95-98% of the requests. In figure 8 (bottom), we can see that after the first two iterations, lookupDN messages are resolved within 4-5 hops. This result shows that resolving lookup in a completely decentralized fashion doesn't actually impose a large overhead of messages even in larger topologies. Another interesting observation is that although the network size was doubled in the LRR-10K dataset, the number of hops taken by the lookupDN message has only slightly increased (≈0.2 messages).

V. Related Work

The need of topologically-aware *unstructured* overlay networks has recently been addressed [19], [14]. In the proposed BinSL algorithm [19], which was evaluated in this work, end-to-end delays are minimized using a system of k landmarks. Recently an approach to create resilient unstructured overlays with small diameters was proposed in [24]. In the proposed algorithm a node selects from a set of k nodes, r nodes at random ($r \subset k$) and then finds from the rest f=k-r nodes the ones that have the largest degree. The algorithm results in networks with powerlaw distributions of node degrees differentiating it therefore from DDNO in which we have a uniform distribution.

Topologically-aware overlays have also been addressed in the context of *Structured* P2P overlays in [4], [19], [25], [29]. These systems rely on some hashing

scheme which allows nodes to quickly send messages to some destination node. Although structured overlays are of particular importance in applications such as decentralized web caches [11], they are not appropriate for content-based retrieval systems [28] and large-scale systems with transient user populations [5]. Li *et al* [17] propose techniques to construct overlay networks (meshes). However, their techniques are not distributed.

Application-layer multicast systems such as Narada [6] initially construct a richer connected graph (mesh) and then use some optimization algorithm to generate a mesh that has certain performance properties. As part of the mesh quality improvement algorithm, Narada nodes randomly probe each other and calculate the perceived gain in utility. We believe that our approach is simpler and cheaper in terms of messages. It is furthermore designated for larger groups of members, which might leave and join in an ad-hoc manner.

Finally, network-awareness is also addressed in the context of large-scale service overlays [12]. In the proposed scheme, a hierarchically fully connected topology of nodes that are clustered based on their distances is constructed. Although the centralized clustering component might be fast and accurate, decentralized approaches are more scalable and less expensive.

VI. Conclusions & Future Work

In this work we propose and evaluate *DDNO (Distributed Domain Name Order)*, which is a distributed technique to make unstructured overlays topologically-aware. We compare DDNO with a number of other overlay construction techniques in both centralized and distributed settings. Our experiments indicate that DDNO is an attractive technique for topologically aware overlay construction as it optimizes many desirable properties such as end-to-end delays, diameter and avoids network partitioning, scales to large overlay networks and works well in dynamic environments.

We believe that our technique is simple enough to be incorporated in existing overlay systems with minimum changes to the respective protocols. In the future we want to test our technique using new datasets, such as a dataset of latencies that we are in the process of obtaining from Akamai. We are further interested in deploying our middleware platform, which is currently under development, in a larger and more realistic simulation over the PlanetLab [18] distributed overlay testbed which is expected to run over 1000 geographically distributed machines in the next few years.

References

[1] Andersen D., Balakrishnan H., Kaashoek F. and Morris R. ”Resilient Overlay Networks”, INFOCOM 2004.

[2] Batagelj V. and Mrvar A. ”PAJEK - Program for large network analysis”, Connections, 21:47–57, 1998.

[3] Bollobás B. ”Modern Graph Theory, Graduate Texts in Mathematics” vol. 184, Springer-Verlag, New York, 1998.

[4] Castro M., Druschel P., Charlie Hu Y., Rowstron A. ”Topology-aware routing in structured peer-to-peer overlay networks”, *In IFIP/ACM Middleware*, 2001.

[5] Chawathe Y., Ratnasamy S., Breslau L., Lanham N., Shenker S. ”Making Gnutella-like P2P Systems Scalable”, *In ACM SIGCOMM'03*, Karlsruhe, Germany, August 2003.

[6] Chu Y-H, Rao S.G., Zhang H. ”A Case For End System Multicast”, *In ACM SIGMETRICS*, 2000.

[7] Crespo A. and Garcia-Molina H. ”Routing Indices For Peer-to-Peer Systems”, *In ICDCS'02*, Vienna, Austria, 2002.

[8] Gnutella, `http://gnutella.wego.com`.

[9] Gross J.L. and Yellen J. ”Graph theory and its applications”, CRC Press, 1999.

[10] Hansen, T., Otero, J., McGregor, A., Braun, H-W., ”Active measurement data analysis techniques”, *In CIC*, June 2000.

[11] Iyer S., Rowstron A., Druschel P., ”SQUIRREL: A decentralized, peer-to-peer web cache” *In PODC 2002*.

[12] Jin J. and Nahrstedt K. ”Large-Scale Service Overlay Networking with Distance-Based Clustering”, *In IFIP/ACM Middleware'03*, pp 394-413, Rio de Janeiro, Brazil, June 2003.

[13] Kazaa, `http://www.kazaa.com/`

[14] Liu Y., Liu X., Xiao L., Ni L. M., Zhang X. ”Location-aware topology matching in P2P systems”, *IEEE INFOCOM 2004*.

[15] Mockapetris P. ”Domain Names - Implementation and Specification”, RFC-1035, Network Working Group, Nov. 1987.

[16] Napster, `http://www.napster.com/`.

[17] Li Z. and Mohapatra P. ”The impact of topology on overlay routing service”, *in IEEE INFOCOM 2004*.

[18] PlanetLab `http://www.planet-lab.org/`.

[19] Ratnasamy S., Handley M., Karp R., Shenker S. ”Topologically-Aware Overlay Construction and Server Selection”, *In IEEE INFOCOM'02*, NY, June 2002.

[20] Ratnasamy S., Francis P., Handley M., Karp R., Shenker S. ”A Scalable Content-Addressable Network”, *In ACM SIGCOMM'01*, August 2001.

[21] Rowstron A. and Druschel P., ”Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems”, *In IFIP/ACM Middleware'01*, Nov. 2001.

[22] Stoica I., Morris R., Karger D., Kaashoek M.F., Balakrishnan H. ”Chord: A scalable peer-to-peer lookup service for Internet applications”, *In ACM SIGCOMM'01*, August 2001.

[23] Subrmanian L., Stoica I., Balakrishnan H., Katz H. R. ”OverQoS: An Overlay based Architecture for Enhancing Internet QoS”, *In INFOCOM 2004*, Hong Kong, March 2004.

[24] Wouhaybi R. and Campbell A. ”Phenix: Supporting Resilient Low-Diameter Peer-to-Peer Topologies” *In IEEE INFOCOM'04*, Hong Kong, to appear in 2004.

[25] Xu Z., Tang C., Zhang Z. ”Building Topology-Aware Overlays using Global Soft-State”, *In ICDCS 2003*, Providence, RI.

[26] Yang B., and Garcia-Molina H. ”Efficient Search in Peer-to-Peer Networks”, *In ICDCS'02*, Vienna, Austria, July 2002.

[27] Zeinalipour-Yazti D. and Folias T., ”Quantitative Analysis of the Gnutella Network Traffic”, Tech. Rep. UC-CS-89, UCR.

[28] Zeinalipour-Yazti D., Kalogeraki V., Gunopulos D. ”Exploiting Locality for Scalable Information Retrieval in Peer-to-Peer Systems”, Information Systems Journal, to appear in 2004.

[29] Zhao B.Y., Duan Y., Huang L., Joseph A.D., Kubiatowicz J.D. ”Brocade: landmark routing on overlay networks” In IPTPS'02, Cambridge MA, March 2002.