Perturbation-Resistant and Overlay-Independent Resource Discovery

Steven Y. Ko and Indranil Gupta
Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, IL 61801
{sko,indy}@cs.uiuc.edu

Abstract

This paper realizes techniques supporting the position that strategies for resource location and discovery in distributed systems should be both perturbation-resistant and overlay-independent. Perturbation-resistance means that inserts and lookups must be robust to ordinary stresses such as node perturbation, which may arise out of congestion, competing client applications, or user churn. Overlay-independence implies that the insert and lookup strategies, and to an extent their performance, should be independent of the actual structure of the underlying overlay. We first show how a well-known distributed hash table (Pastry) may degrade under perturbation. We then present a new resource location and discovery algorithm called MPIL (Multi-Path Insertion/Lookup) that is perturbation-resistant and overlay-independent. MPIL is overlay-independent in that it effectively provides to the distributed application an ability to insert and lookup Pastry objects in an overlay with Pastry IDs, but without the need to have Pastry-style overlay maintenance (i.e., the overlay underneath can be arbitrary). We quantify, through analysis and simulation results, the behavior of MPIL over complete, random, and power-law overlays. We also show how MPIL outperforms regular Pastry routing when there is perturbation.

1 Introduction

Resource location and discovery in distributed systems such as the Grid, cooperative web caching, peer to peer email, etc., all require object insertion and querying mechanisms that are scalable and tolerant to node failures. This paper is motivated by two additional practical concerns that require far more from such object insertion and querying strategies (henceforth, together labeled as "lookup" strategies). These practical concerns are *overlay-independence* and *perturbation-resistance*.

Lookup strategies are usually coupled with the maintenance of an appropriately matched application-layer network ("overlay") among the participating hosts ("nodes" or "peers") on top of the Internet. Each node knows a few other nodes in the overlay according to specific overlay rules, and routes overlay messages such as insertion and querying of files. However, this makes it impossible to deploy a practical peer to peer application (e.g., cooperative web caching) on an *already-existing legacy overlay* (e.g., a Grid network) without first deploying the overlay maintenance protocols associated with the p2p application. These maintenance protocols might increase the overhead, or worse inhibit the performance of, other already-existing protocols in the legacy overlay that already maintain some kind of structure. This motivates the need to develop lookup strategies that are independent of the structure of the underlying overlay, and perform well under various arbitrary overlay topologies.

A second and more important practical concern is perturbation-resistance. If p2p overlays are to be deployed successfully for a variety of legal applications, the robustness of their behavior under the ordinary kinds of stress experienced by nodes will be a minimum requirement. Perturbation is one such kind of stress. A node is said to be *perturbed* if it is unresponsive for brief periods of time. Perturbation can be caused by many reasons and can occur at several granularities. Concurrent competing applications running on the host, packet buffer overflows, and congestion, can cause short-term perturbation, where the node is unresponsive for up to a few seconds. Longer-term perturbation with unresponsiveness granularities of several minutes or hours can be caused by user churn, i.e. rapid node departures and arrivals of users, a phenomenon present in Grid applications and file sharing overlays. In this paper, we model perturbation by nodes whose availability *flaps* periodically, and study the effect of such periodic flapping on the lookup success rate. Success rate is the fraction of successful replies to lookups injected into the overlay.

Currently, overlays are either unstructured or structured. Unstructured overlays such as Gnutella [1] use flooding to query object replicas. While this strategy is perturbation-resistant and overlay-independent, it is neither efficient nor

scalable. Structured overlays include Chord [17], Pastry [15], Tapestry [18], Kelips [7], Viceroy [11]. Also called DHTs (Distributed Hash Tables), these overlays map both objects and nodes to keys by using hash functions. Lookups are then routed within this overlay by using a routing algorithm that selects *one* next hop node at each step, based on key values of the destination and the current node. While structured overlay lookups are efficient and scalable, they are not overlay-independent because the routing algorithm is usually coupled with an appropriate overlay structure with maintenance strategies. For example, Pastry uses prefix routing based on key values, and nodes in the underlying overlay select neighbors based on the same metric. Recent studies reveal that many structured overlays may be churn-resistant, but we show in this paper that they may not be resistant to more general perturbations.

We present a new resource location and discovery algorithm called MPIL (Multi-Path Insertion/Lookup) that provides both overlay-independence and perturbation-resistance. MPIL achieves these goals by using a deterministic routing metric (like DHTs), but by exploiting *limited* redundancy (like unstructured p2p systems). The deterministic routing metric used is based on the hash value of keys (objects and nodes), just like Pastry or Chord, but unlike those systems, does not assume any characteristics about the underlying topology. This routing requires the use of limited redundant routing of lookups to insert and query multiple replicas of an object pointer. This limited redundant routing also provides perturbation-resistance. Put together, MPIL provides a cost-effective and convenient way of developing and deploying robust p2p applications that target any type of overlay. In a sense, the techniques of limited redundancy and overlay-independence achieves the best of both worlds from both structured and unstructured overlays.

The rest of this paper is organized as follows. We present the related work (Section 2), the effect of perturbation on *MSPastry*, the original implementation of Pastry[1] (Section 3), the MPIL algorithm (Section 4) and its analysis (Section 5), simulation results (Section 6), and conclusion (Section 7).

2 Related Work

Perturbation has been studied in other contexts besides overlay networks. Birman *et al.*[3] study the effect of perturbation in the context of multicast protocol. In their simulation, virtually synchronized multicast groups are used to study the effect of perturbation. They measure throughput of a live node in the presence of perturbation - some fraction of the multicast group members sleep for some fraction of each second. Their result shows that even with a single perturbed group member, the throughput drops significantly, decreasing rapidly as the number of perturbed nodes increases.

Many studies have shown that the arrival rate and the departure rate of nodes in peer-to-peer systems are very high, which proves the instability of peer-to-peer systems. For example, Bhagwan *et al.*[2] show churn data for the Overnet p2p system. Saroui *et al.* [16] study node availability of Napster and Gnutella. Their result can be summarized as the best 20% of Napster peers has an uptime of 83% and more, and the best 20% of Gnutella peers has an uptime of 45% or more.

Robustness issues of DHTs have been studied recently [6] [14] [9]. Li *et al.*[9] study the effect of churn to some popular DHTs including Chord, Tapestry, Kelips, and Kademlia. Castro *et al.*[6] study the performance and dependability of Pastry with MSPastry implementation. Rhea *et al.*[14] identify three factors that affect the behavior of overlays under churn - reactive vs. periodic recovery, message timeout calculation, and proximity neighbor selection - and discuss various techniques that can be used for the three factors.

Efficient search algorithms for unstructured overlays have been studied recently [10] [13]. Lv *et al.*[10] explore the use of random walks and replication to improve the inefficiency of unstructured p2p systems caused by flooding. More recently, it has come to our attention that Morselli *et al.* [13] propose a search algorithm that combines random walks and a DHT routing algorithm based on Chord routing algorithm to improve the search efficiency of unstructured p2p systems. This study shares some similarities with our study in that 1) it uses name-space virtualization for unstructured overlays, 2) it proposes a replication strategy, and 3) it separately considers overlays and routing algorithms. However, their focus is producing unstructured p2p systems, while our focus is producing robust p2p systems.

Castro *et al.* [5] use flooding and random walks over Pastry's structured overlay to support complex queries. Our approach is different, because we develop a robust resource discovery algorithm that runs over any type of overlay.

3 Effect of Perturbation on Pastry

To study the effect of perturbation on Pastry, we conduct a set of simulations with MSPastry. Our result indicates that although MSPastry already has various overlay maintenance techniques that deal with failures in overlays, they are not sufficiently perturbation-resistant.

Figure 1 summarizes our results. Each simulation consists of two stages. In the first stage, 1000 insertion requests are generated to the *static* overlay of MSPastry. These 1000 insertion requests have randomly-generated unique message IDs. In the second stage, 1000 lookup requests are generated by the same node which generates the insertion requests in the first stage. The lookup requests are gener-

[1] Obtained under a limited license from Microsoft Research.

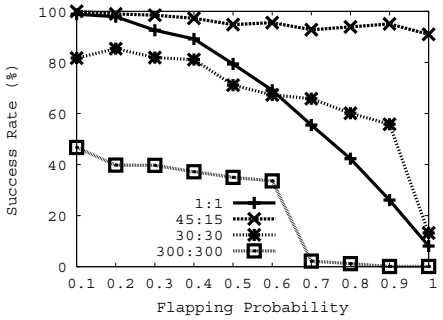

Figure 1. *The effect of perturbation on MSPastry. x-axis (flapping probability) indicates the probability for a node to get perturbed. 1:1 indicates that the online period is 1 second and the offline period is 1 second. The same goes for 45:15 (idle:offline=45:15), 30:30 (idle:offline=30:30), and 300:300 (idle:offline=300:300).*

ated every (online period + offline period) seconds one by one. These 1000 lookup requests are the lookup requests for 1000 IDs inserted in the first stage. The overlay in the second stage is not static; Each node gets perturbed with some probability. As mentioned earlier, our model of perturbation can be described as *flapping*. A perturbed node periodically flaps between being offline and being idle (online). At the beginning of each idle period, every node comes back online and stays online during the period. At the beginning of the offline period, however, each node decides whether to go offline or to stay online based on the flapping probability. Each node randomly picks its very first beginning of the flapping period (i.e. idle period + offline period). Lookups are performed after every node enters its flapping period.

As in Figure 1, when idle:offline period is 45:15 (seconds), MSPastry can route more than 90% of the messages successfully. This result shows that MSPastry is already robust to a certain level, which is due to the overlay maintenance techniques of MSPastry. However, the number of successful lookups decreases in other cases and with higher flap rates in general. When idle:offline period is 30:30 (seconds), the success rate is roughly about 85% even with the flapping probability of 0.1. When idle:offline period is 1:1 (seconds), the success rate drops almost linearly. With idle:offline period of 300:300 (seconds), the success rate is almost 0 with the flapping probability from 0.8 to 1. This result clearly shows that the overlay maintenance techniques of MSPastry are perturbation-resistant only to a limited degree. Further, both short-term perturbations (e.g., 1:1) and long-term perturbations (e..g, 300:300) drastically affect the lookup behavior.

4 Multi-Path Insertion/Lookup Algorithm

Figure 2 shows the overall architecture of MPIL (Multi-Path Insertion/Lookup). The insertion and lookup operations use the routing algorithm, and the routing algorithm rely on two important bases, a routing metric and a traffic

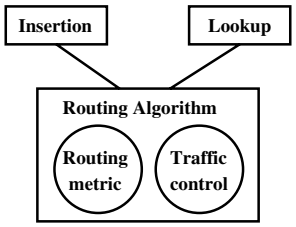

Figure 2. *The architecture of MPIL. Insertion and lookup operations use the routing algorithm, and the routing algorithm uses a routing metric and a traffic control algorithm.*

Figure 3. *An example for the routing metric. On the Left, ID 1001 and 1011, on the right, ID 1001 and 0010. The routing metric gives the value of 3 and 1, respectively.*

control algorithm. In fact, the insertion and lookup operations become straight-forward once we understand the routing algorithm.

The MPIL routing algorithm works as follows: when a node receives a lookup request for an object, it calculates the routing metric for each of its neighboring peers, and forwards the lookup to the "best" few peers. Below, we first describe how the routing metric is calculated for a given object, and then detail the routing algorithm itself.

4.1 Routing Metric

For a given object ID and a neighboring peer's ID, the routing metric is simply the number of matching digits appearing in same positions. Another way to view this metric is the number of 0's in XOR product of the two ID's; this is related to the concept of Hamming distance. Unlike the Kademlia overlay [12], which also uses an XOR, MPIL uses the XOR metric to select multiple next hops for the query – we detail this in Section 4.2.

Figure 3 shows an illustration of this routing metric. Consider the nodes with id's 1001 and 1011 from the 4-bit ID space (example on the left). The value of the MPIL routing metric is 3, since only the second-least significant bits do not match. Suppose a node currently holds a lookup request for an object with ID 1001, and the node has two neighbors 1011 and 0010. Since the values returned by MPIL are 3 and 1 respectively, the lookup is forwarded to node 1011.

4.2 Properties of MPIL's Routing Metric

The routing metric of MPIL has the following advantages over other routing metrics that exist for structured overlays.

Continuous Forwarding over Arbitrary Overlays MPIL is better than prefix or suffix routing at distinguishing neighbors of a node when trying to select the best next hop for a lookup message. For example, in the overlay of Figure 4, if node 1001 has a lookup message for

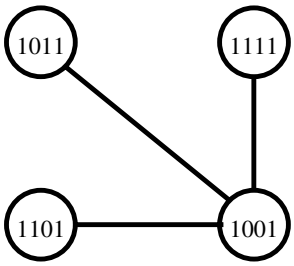

Figure 4. *An example topology for continuous forwarding*

object 0110, both prefix routing and suffix routing treat all neighbors as being equivalent - this may cause the lookup message to be dropped or evaluated again to break the tie. However, the MPIL routing metric returns 1111 as the best neighbor.

The reason can be explained probabilistically. In prefix routing, the probability that any given two IDs share no common prefix at all is 0.75 for base-4 representation, and 0.5 for binary representation. Considering that having a common prefix is a basic requirement, the probability needs to be far lower than 0.75 or 0.5 in order for the prefix routing to be used over arbitrary overlays. This problem becomes worse especially when the large fraction of the total nodes has only a small number of neighbors, e.g. power-law graphs.

On the other hand, for the MPIL routing metric, the probability of the above event is only $(\frac{3}{4})^{80} = (1.0113490...)^{-10}$ if we assume 160-bit ID space and base-4 representation. The MPIL routing metric thus distinguishes neighbors better; at the least, this ensures a lookup request undergoes *continuous forwarding* even over arbitrary overlays.

Redundancy For Robustness The MPIL routing metric provides an easy way to exploit redundancy for robustness since it provides an inherent way to create multiple paths to multiple peers. Since the MPIL routing metric counts the number of common digits in same positions, there can be multiple nodes that have the same number of common digits. In Figure 4, suppose the node 1001 forwards a message of ID 0001. 1111 and 0001 share 1 common digit, 1101 and 0001 share 2 common digits, and 1011 and 0001 share 2 common digits. Thus, 1101 and 1011 are both the candidates for the next hop. Unlike other routing algorithms that break this tie using other mechanisms, MPIL forwards messages to every candidate, thus creating multiple paths to multiple peers. We use the term *flows* or *paths*. If a node forwards a query to exactly one neighboring node, there is only one flow. For each additional neighbor that is chosen to forward the lookup, an additional flow is said to be created.

Such replication might cause some nodes to receive the same message. In this case, there are two options. A node can either silently discard the message and not forward it any more (thus stopping the flow), or forward the message again. We explore both options in our simulations.

The effectiveness of such redundancy is limited for prefix and suffix routing due to the lower distinguishability of their routing metrics.

4.3 MPIL Routing

MPIL routing works as follows; When a node receives a lookup message containing an object ID, the node calculates the value of each neighbor's routing metric w.r.t. this object, as described in Section 4.1. The node then forwards the message to the neighbor having the highest value. In the case that the node has several neighbors with the same highest value, the node has to choose multiple nodes from among all such highest-value neighbors.

To prevent message explosion, a message field called max_flows is used to limit the number of extra flows created. max_flows is an integer field in every message, and it is decreased each time a node creates an *additional* flow (recall that forwarding to exactly one node is not considered as an additional flow). When max_flows is decreased to 0, no additional flows can be created. This max_flows is conceptually similar to "quota" that is consumed by each node on a route whenever a node replicates a message. The original max_flows value of a message is specified by the originator of the message.

To summarize, when a node receives a message, it does the following:

1. Creates a list of possible candidates for forwarding the message.
2. Compares the size of the list and $(max\_flows + given\_flows)$, where $given\_flows$ is 0 if the node is the original sender, and 1 otherwise. max_flows is specified in the message.
3. Picks the minimum value of the two (say m).
4. Forwards the message to m nodes from the candidate list.
5. Replace the value of max_flows of each message that the node forwards to $(max\_flows - m + given\_flows)/m$.

In the last step, the decrease of max_flows by $m - given\_flows$ is because that is the number of *additional* peers that the node forwards the message to. The node divides the value by m for distribution of the original max_flows. If the final value is not an integer, a node can distribute the residue one by one in round-robin fashion to the m nodes.

The complete MPIL algorithm uses the routing metric in Section 4.1 and the algorithm for limiting multiple flows. Figure 5 shows the pseudo-code of the algorithm. Note that when choosing *next_hop_list* from *neighbor_list*, the number of common digits between *N* and *M* does not have any effect. In addition, there is a message field called *route*, which contains the list of nodes that the message has visited. The

```
M = Message ID
N = Node ID

if M has been forwarded already, discard it (optional).
for node in (neighbor_list of N - M.route):
    C = common digits between M and node
    if C is the largest until now:
        next_hop_list = [node]
    elif C is equal to the largest until now:
        next_hop_list.add(node)
Count common digits between M and N
if N has the largest value among all nodes in neighbor_list:
    N is the destination
    Perform message-specific actions (for insertion messages)
else:
    Apply the paths-limiting algorithm to next_hop_list
    Forward to nodes in next_hop_list
```

Figure 5. *A Pseudo-code of the MPIL Routing Algorithm*

route field prevents the message from being forwarded to a node the message has visited already. Thus, Choosing *next_hop_list* is dependent only on peers in *neighbor_list*, excluding the nodes in *M.route* and *N*. Depending on the configuration, each node might discard a message that has been forwarded already. In this case, a sequence number or a random number should be attached to distinguish the message from old messages with the same message ID.

4.4 Insertion, Lookup, and Deletion

Both insertion and lookup use the routing algorithm, but each has differences in its specifics. We discuss each in detail.

Insertion An object (or a pointer to its location) can be inserted using MPIL routing. An insertion message is propagated and replicated as usual in the MPIL routing algorithm, and an object is inserted at a node when none of its neighbor nodes have a higher MPIL routing metric value than the node. We call such nodes as *local maxima*. This results in multiple replicas of the object being created.

Replica placement is done by specifying the number of per-flow replicas (we call this number *num_replicas* for the discussion. In Figure 5, it says that if *N* is the destination, then it performs message-specific actions. In the case of an insertion message, *N* stores the object location specified in the message. However, this process continues *num_replicas*-times to store more replicas. This is possible because each node picks the next hop from (*its neighbor_list - M.route*) as in Figure 5, a list that does not include the node itself.

Since *max_flows* is the maximum number of possible paths, and each path creates *num_replicas* replicas, the maximum total number of replicas created by an MPIL object insertion request is bounded from above by *max_flows* $\times$ *num_replicas*.

Querying A lookup message that is a query for an object is propagated in essentially the same manner as insertion requests above. However, each recipient node checks to see it has the object; if it does, it stops forwarding the query and

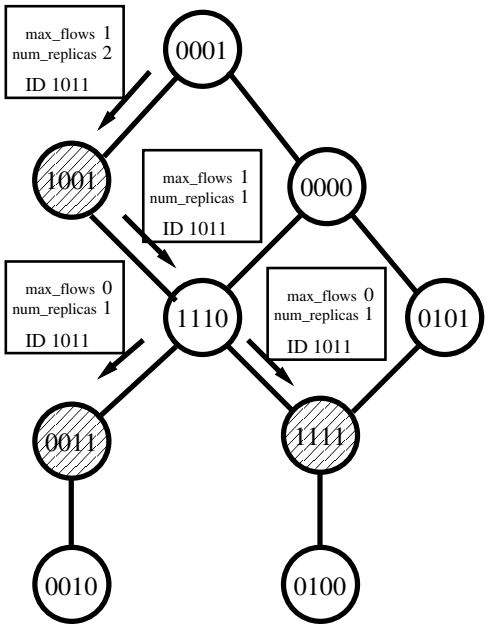

Figure 6. *An example of MPIL. Gray nodes store the location of the object.*

replies back directly to the querying node. The forwarding process stops when either the location is found or the message has passed through *num_replicas* local maxima. For arbitrary overlays, although MPIL can never guarantee a 100% lookup success rate, our simulations reveal that the success rate of MPIL are close to 100%.

Deletion Deletion can be done in many ways, but here we discuss just one of them. Whenever a replica is placed in a node, the node sends a periodic heartbeat to the owner of the original object. When the originator wants to delete a replica, it sends an explicit delete message to the node.

4.5 Comprehensive Example

Figure 6 shows an example of how MPIL inserts and queries an object. Suppose the node 0001 wants to insert an object with ID 1011. Originally, *max_flows* is 2 and *num_replicas* is 2. After node 0001, *max_flows* becomes 1. The node 0001 first selects 1001 among its neighbors because 1011 and 1001 share three common digits, while 1011 and 0000 share only one digit. Since 1001 shares the largest common digits among all of its neighbors, 1001 stores this object and decrements *num_replicas* by 1. But it still forwards the message to 1110 since *num_replicas* is 1. Since 1110 has two neighbors that share three common digits and *max_flows* is still 1, 1110 forwards this message to both neighbors. 0011 and 1111 receive this message and store the location because they share the largest common digits among all of their neighbors. They stop forwarding because *num_replicas* has reached 0. Lookup messages follow the exact same steps, but every node along the routes checks if it has the location. The notion of *flow* can be explained using Figure 6 again. There are two flows. One is

from 0001 to 0011, and the other is from 0001 to 1111. We say that one additional flow is created by 1110.

5 Analysis

In this section, we present the analysis results of MPIL over various types of overlay topologies. First, we study the expected number of local maxima (See Section 4.4 for the definition of local maxima), which is an upper bound on the expected number of replicas, and expected number of hops to a local maximum from a node in general topologies. Recall that a local maxima node may store a replica (if the insert message reaches the node). We study two examples, random regular topologies and complete topologies.

There are several assumptions for the analysis. We assume that 1) there is an m-bit ID space with base-2^b representation, where $m = Mb$ for some constant M. Thus, each ID is a M-character-wide string with 2^b possible characters. 2) the total number of nodes is N, and the degrees is d. 3) There is a message with ID a, and node IDs are $a_0, \ldots, a_N$. We say that a node ID, a_i, is *k-common* when a_i shares k common digits with the message ID, a.

5.1 General Overlay Topologies

For the expected number of local maxima, assume that the degree distribution function of a given type of overlays is known. Then, we can calculate the expected number of local maxima in an overlay topology by $N \times C$, where

$C = \sum_{d=1}^{N} \left\{ P(\#\text{ of neighbors} = d) \sum_{k=1}^{M} \left(A \times B^d\right) \right\}$

$A = \binom{M}{k} \left(\frac{1}{2^b}\right)^k \left(\frac{2^b-1}{2^b}\right)^{M-k}$

$B = \sum_{j=0}^{k-1} \binom{M}{j} \left(\frac{1}{2^b}\right)^j \left(\frac{2^b-1}{2^b}\right)^{M-j}$

C is the probability for a node to become a local maximum. A is the probability for a node to be *k-common*, and B is the probability for every other node to be *j-common*, for some $j < k$.

If we assume that the local maxima are distributed uniformly over the topology and we perform a random walk over the topology, then the expected number of hops to reach one of the local maxima from any node in the overlay is simply $\frac{1}{C}$.

Thus, if the degree distribution function is known, we can calculate the expected number of local maxima and the expected number of hops to one of the local maxima.

5.2 Random Regular and Complete Topologies

The degree distribution function of random regular overlay topologies, where each node has fixed d neighbors is given by,

$$P(\#\text{ of neighbors} = i) = \begin{cases} 1 & \text{if } i = d \\ 0 & \text{otherwise} \end{cases}$$

Then, we can calculate the average number of local maxima in a random regular topology, which is $N \times C$, where $C = \sum_{k=1}^{M} \left(A \times B^d\right)$. C is a constant for a given d. Figure 7 shows the average number of local maxima with different number of nodes and neighbors.

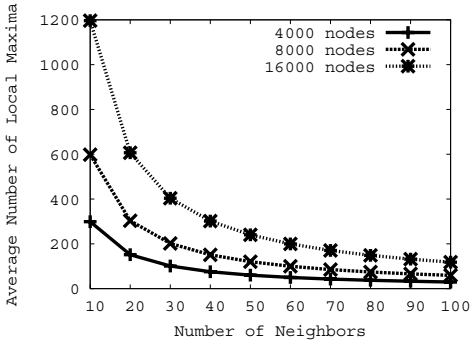

Figure 7. *The expected number of local maxima for random regular topologies*

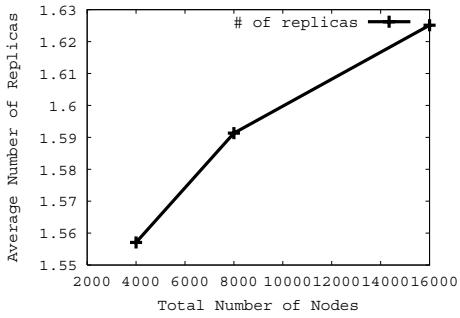

Figure 8. *The expected number of replicas for complete topologies*

The expected number of hops to one of the local maxima by a random walk is also a constant, $\frac{1}{C}$

Similarly, we can calculate the expected number of replicas in a complete topology by using a similar equation, which is, $N \times \sum_{k=1}^{M} \left(A \times D^{N-1}\right)$, where

$D = \sum_{j=0}^{k} \binom{M}{j} \left(\frac{1}{2^b}\right)^j \left(\frac{2^b-1}{2^b}\right)^{M-j}$.

Compared to the equation for random topologies, this equation uses $N-1$ instead of d, since it assumes a complete topology. Also, D is exactly the same as B except that the summation includes k, since it considers the number of replicas. Figure 8 shows the results for various number of nodes.

6 Simulation Results

Two different classes of simulations are performed to examine MPIL and its robustness. The first class of simulations is mainly to evaluate the MPIL insertion/lookup performance over various static overlays. We wrote an application message simulator in Python for overlay-level routing. The second class of simulations evaluates the robustness of MPIL over structured overlays using MSPastry [6]. For ID-generation, we use random numbers picked from 160-bit ID space. All simulations are done on Pentium4 2.7GHz and 512M RAM.

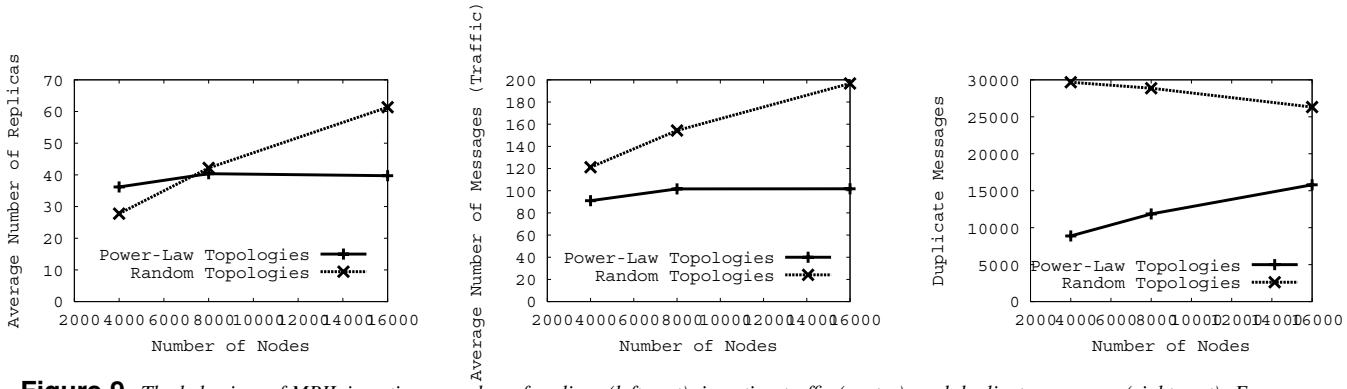

Figure 9. *The behaviors of MPIL insertion - number of replicas (leftmost), insertion traffic (center), and duplicate messages (rightmost). Even though the plots seem to increase, they are actually bounded by max_flows and per-flow replicas specified by the originator, regardless of the number of nodes. For example, the number of replicas is limited by max_flows × per-flow replicas, which is 150 in the simulations.*

6.1 MPIL Insertion/Lookup Performance over Static Overlays

Overlay Topologies There are very few reliable benchmarks or workload generators for legacy distributed applications like Grid applications. However, we believe that power-law or random graph structures may be natural for legacy application overlays; we use these below.

10 different power-law graphs are generated by Inet [8], each with 4000 nodes, 8000 nodes, and 16000 nodes. We use 0% of degree 1 nodes. Similarly, 10 different random graphs are generated, each with 4000 nodes, 8000 nodes, and 16000 nodes. In these random graphs, each node has 100 neighbors, equally.

Methodology For each overlay, random nodes are chosen to insert objects with different IDs 100 times. After that, those 100 objects are queried one by one again by randomly chosen nodes. Since there are 10 different overlays for each 4000 nodes, 8000 nodes, and 16000 nodes, the total number of insertion/lookup pairs is 1000 for every number of nodes. For all insertions and lookups, a node silently discards a message if the node receives the same message more than once.

Insertions Figure 9 shows the insertion performance of MPIL over the power-law and random overlays. The maximum number of flows is fixed at 30 and the per-flow replicas is fixed at 5 (See Section 4.2, 4.3, and 4.4 for the notion of flows, maximum flows specified by originators, and per-flow replicas). We measure two different categories - number of replicas and traffic per insertion - over 4000 nodes, 8000 nodes, and 16000 nodes.

The leftmost graph of Figure 9 shows the average number of replicas per insertion and the center graph of 9 shows the average number of total messages per insertion. The rightmost graph of Figure 9 shows the total number of duplicate insertion requests. Whenever a node receives the same insertion request from a different neighbor, it is considered as a duplicate request.

The number of replicas is bounded by (the maximum number of flows) × (the number of per-flow replicas). Thus, in the leftmost graph of Figure 9, the maximum number of replicas is bounded from above by 150, regardless of the number of nodes. As discussed earlier, additional flows are not created at every node, but at a node that has multiple neighbors with the same number of common digits. Thus, the actual number of flows is usually less than the maximum specified by the originator. The leftmost graph of Figure 9 shows this behavior as well. Even with 30 maximum flows and 5 per-flow replicas, the actual number of replicas is much less than 150.

A couple of further observations can be made from Figure 9. First, the number of replicas and traffic of insertions in the power-law overlays stay almost the same across different settings. The reason can be found in the rightmost graph of Figure 9. As the number of nodes increases, the number of duplicate messages increases in the power-law overlays. This means that more duplicate messages arrive at the same set of nodes and are silently discarded, which prohibits an insertion from storing more replicas. Second, the number of replicas and traffic increases in the random overlays in contrast to power-law overlays. The reason can be found again from the duplicate messages. In the rightmost graph of Figure 9, the number of duplicate messages decreases as the number of nodes increases in the random overlays. Thus, more messages follow different paths as the number of nodes increases, which leads to storing more replicas.

The difference in number of duplicate messages between random overlays and power-law overlays is because each node has 100 neighbors in the former, while many nodes have only a few neighbors in the latter.

Lookups Table 1 and 2 show the success rates of MPIL lookups in various settings. Note that the per-flow replicas (r) for lookups means that the lookup stops when a flow encounters a node with the largest common digits r-times. Insertions are performed before lookups, and the number of maximum flows is fixed at 30 and the number of per-flow replicas is fixed at 5. Since we consider insertions to be rare events compared to the lookups, the traffic of insertions

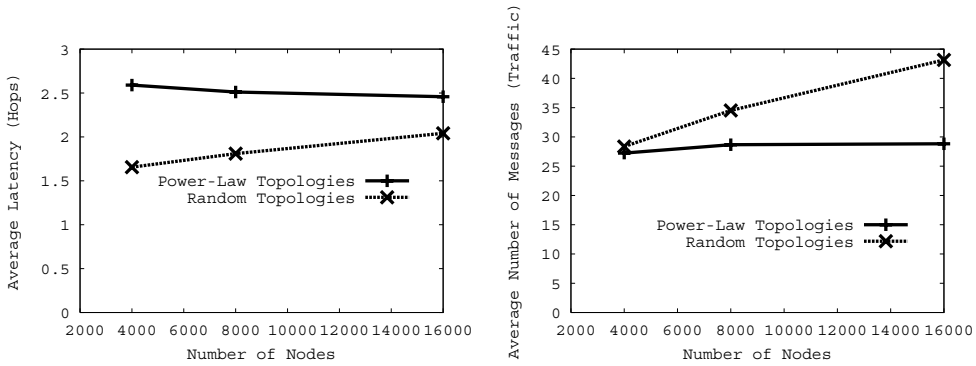

Figure 10. *MPIL Lookup - latency (hops) (left), lookup traffic (right). Even though the traffic seems to increase, it is actually bounded by max_flows and per-flow replicas specified by the originator, regardless of the number of nodes.*

Table 1. *MPIL lookup success rate over power-law topologies*

		Per-flow Replicas				
# nodes	Max_flows	1	2	3	4	5
4000	5	52.9	94.4	97.7	98.7	99.1
	10	55.4	98.7	99.7	99.9	100
	15	56.0	99.0	99.7	99.9	100
8000	5	57.1	96.5	98.8	99.6	99.2
	10	60.5	99.2	100	100	100
	15	60.0	99.6	100	100	100
16000	5	58.3	98.1	99.7	99.9	99.9
	10	60.4	99.5	100	100	100
	15	60.9	99.8	100	100	100

Table 2. *MPIL lookup success rate over random topologies*

		Per-flow Replicas				
# nodes	Max_flows	1	2	3	4	5
4000	5	98.6	100	100	100	100
	10	98.8	100	100	100	100
	15	98.4	100	100	100	100
8000	5	97.0	99.9	100	100	100
	10	98.5	100	100	100	100
	15	98.7	100	100	100	100
16000	5	95.0	99.9	100	100	100
	10	98.4	100	100	100	100
	15	98.6	100	100	100	100

caused by the large number of maximum flows and per-flow replicas can be amortized over time.

From Table 1 and 2, three observations can be made. First, having more per-flow replicas gives higher success rates. This is obvious because the number of per-flow replicas for a lookup limits the path-length of the lookup. Second, having more flows gives higher success rates. This is also obvious because the number of maximum flows limits the number of search paths. Third, having larger numbers of nodes gives higher success rates with the same number of max_flows and per-flow replicas, although the difference is very small and may be negligible. The reason of the difference can be found in Table 3. Table 3 shows an example of the average number of flows that are *actually* created by lookups with 10 max_flows and 3 per-flow replicas over various numbers of nodes. As the number of nodes grows, the actual number of flows grows also, even though the maximum flows and per-flow replicas are the same. Since there

Table 3. *Actual number of flows of lookups*

# of Node	Actual # of Flows
Power-Law 4000	8.782
Power-Law 8000	9.151
Power-Law 16000	9.542
Random 4000	9.323
Random 8000	9.505
Random 16000	9.63

are more flows for bigger overlays, the success rates increase, accordingly.

Figure 10 shows the latency and required traffic of lookups. In this simulation, max_flows is fixed at 10 and per-flow replicas is fixed at 5, since that setting gives 100% success rates for all 4000, 8000, and 16000 nodes in both the power-law overlays and random overlays. Also, the left graph of Figure 10 only shows the number of hops of the first successful reply of a lookup among all successful replies. Multiple successful replies are possible because there are multiple replicas stored in the system. However, the right graph of Figure 10 shows the total traffic per a lookup request, as well as the traffic for the first successful reply.

As in the left graph of Figure 10, the latency stays almost same even though the number of nodes increases. Table 1 shows a similar result because more than 50% of lookups are satisfied even with 1 per-flow replicas and almost all lookups are satisfied with 2 per-flow replicas. Although limiting per-flow replicas does not accurately limit the number of maximum hops a lookup request is propagated, it definitely has a correlation. Therefore, both the left graph of Figure 10 and Table 1 tell us that MPIL lookups cause small and steady number of hops across different numbers of nodes in the power-law overlays.

Lookup traffic also stays almost same in the right graph of Figure 10. The same reason from the case of insertions can be applied to here. Since the number of duplicate messages increases as the number of nodes increases, more flows and traffic are suppressed.

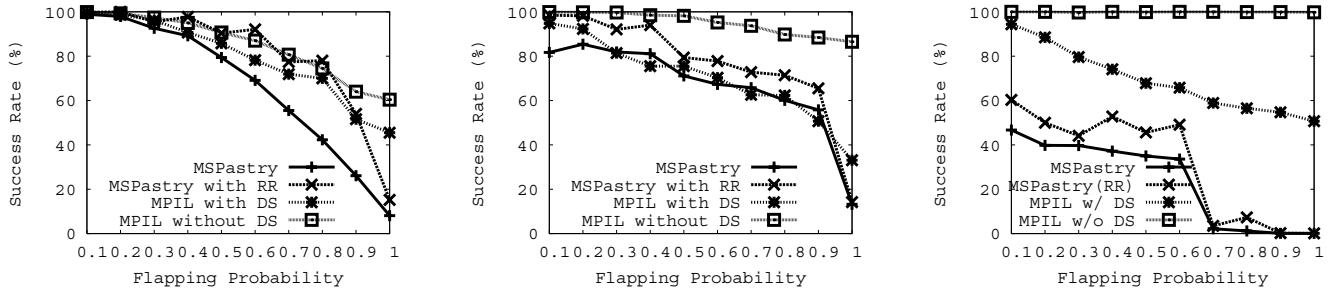

Figure 11. *Success rate of MSPastry simulations - success rate of idle:offline=1:1 (leftmost), success rate of idle:offline=30:30 (center), success rate of idle:offline=300:300 (rightmost) . "MSPastry" is the simulation result using original MSPastry with various overlay maintenance techniques. "MSPastry with RR" is the simulation results using original MSPastry with overlay maintenance techniques plus RR (Replication on Route). "MPIL with DS (Duplicate Suppression)" and "MPIL without DS" are the simulation results using MPIL without overlay maintenance techniques. MPIL without DS gives the best robustness under perturbation.*

6.2 MPIL over MSPastry

In this section, we study the robustness of MPIL under perturbation. We run MPIL over the overlay of MSPastry by implementing the MPIL algorithm in MSPastry. We compare the robustness of MPIL to that of MSPastry with its overlay maintenance techniques.

Methodology To evaluate the robustness of MPIL, we conduct a set of simulations using MSPastry. These simulations are done in the same condition as in Section 3. 1000 insertions are generated first, and 1000 lookups for the same IDs of insertions are generated next. For MSPastry simulations, we use MSPastry with all the overlay maintenance techniques described in [6]. For MPIL simulations, we modify MSPastry; we replace the original routing algorithm of MSPastry with MPIL. However, we do not use any of the overlay maintenance techniques. In other words, we use the structured overlay of MSPastry, but none of the overlay maintenance techniques. A total of 1000 nodes are used in all simulations using a topology generated by GT-ITM [4] as an underlying (Internet) topology.

MSPastry Configuration The default parameters of MSPastry simulations across all simulations are; b = 4, l = 8, *leafset probing period* = 30 seconds, *routing table maintenance period* = 12000 seconds, *routing table probing period* = 90 seconds, *probe timeout* = 3, and *probe retries* = 2.

MPIL Configuration All MPIL simulations are done with 10 maximum flows and 3 per-flow replicas for both insertions and lookups. However, the number of replicas actually inserted by the insertions in the network is typically 6-7.

Success Rate Figure 11 shows the success rates of the original MSPastry and MPIL under various perturbation probabilities. In Figure 11, "MSPastry" shows the simulation results with the original MSPastry with no modification. "MSPastry with RR" shows the results with MSPastry with *Replication on Route* (RR). Using RR, every node on the route of an insertion message stores a replica whether it's the target node or not. In these simulations, the typical number of hops of an insertion message is 2-3 for MSPastry. Thus, 2-3 is the typical degree of replication for MSPastry with RR. "MPIL with DS" shows the simulation results of MPIL with Duplicate Suppression (DS), while "MPIL without DS" shows the results of MPIL without DS. If MPIL uses DS, each node silently discards any message that the node forwarded before. Otherwise, each node forwards a message repeatedly, even if it received the same message before.

Intuitively, DS is good for static overlays because it reduces traffic. However, each node is likely to have a different set of neighbors in dynamic overlays. Thus, if a node keeps forwarding a message in dynamic overlays, the message is likely to take a different route each time, and the chance of arriving at one of the replicas can be increased. Figure 11 actually confirms this intuition. MPIL without DS always gives higher success rates than MPIL with the duplicate suppression. However, MPIL typically gives higher success rates over the original MSPastry, regardless of DS.

Traffic MPIL gives better success rates under perturbation as in Figure 11. However, since MPIL uses multicasts, the traffic generated by MPIL can be far more than that of MSPastry. The left graph of Figure 12 compares the lookup traffic of the original MSPastry and MPIL, when idle:offline is 30:30. As shown, MPIL creates a lot more lookup traffic than the original MSPastry, especially under low perturbation probabilities.

However, the original MSPastry uses various overlay maintenance techniques that create consistent background traffic, while MPIL does not use any of the techniques. The right graph of Figure 12 shows the total number of messages sent including all the maintenance and control messages. As shown, MPIL creates far less messages than the original MSPastry if all the messages are counted. However, if the lookup frequency were higher, MSPastry's overhead might be justified.

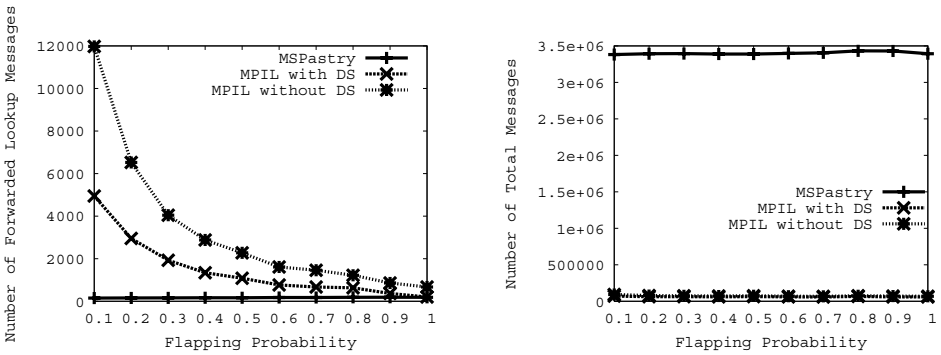

Figure 12. *Overall traffic of MSPastry simulations - lookup traffic (left), total traffic including maintenance messages (right). Two figures show the lookup traffic and overall traffic. MSPastry has constant background traffic of overlay maintenance techniques, while MPIL has more traffic of lookup, but negligible background traffic. Again, we use MPIL with/without DS (Duplicate Suppression).*

7 Conclusion and Future Work

This paper has presented a new approach to resource location and discovery that is both perturbation-resistant and overlay-independent. Our algorithm, called MPIL, is independent of the underlying overlay structure. MPIL works well over both unstructured overlays that are random or power-law, and over structured overlays – like MSPastry. Under both short-term and long-term perturbation (which may arise from multiple concurrent client applications or churn respectively), MPIL has a better success rate than MSPastry routing, and at the cost of only slightly increased communication for each object lookup request. MPIL successfully provides any distributed application the ability to insert and query objects reliably and robustly over any arbitrary overlay, without the need to change the existing overlay maintenance mechanisms.

Acknowledgments We thank the Pastry team at Microsoft Research Labs (Cambridge, UK) for providing us with the MSPastry code and license.

References

[1] The Gnutella protocol specification v 0.4, document revision 1.2. *www.clip2.com*, 2003.

[2] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *International Workshop on Peer-to-Peer Systems*, February 2003.

[3] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 2002.

[4] K. Calvert, M. Doar, and E. W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 1997.

[5] M. Castro, M. Cost, , and A. Rowstron. Should we build Gnutella on a structured overlay? In *Hot Topics in Networks*, November 2004.

[6] M. Castro, M. Cost, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *International Conference on Dependable Systems and Networks*, June 2004.

[7] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: building an efficient and stable P2P DHT through increased memory and background overhead. In *International Workshop on Peer-To-Peer Systems*, March 2002.

[8] C. Jin, Q. Chen, and S. Jamin. Inet: Internet topology generator. *http://topology.eecs.umich.edu/inet*, 2002.

[9] J. Li, J. Stribling, T. M. Gil, R. Morris, and F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *International Workshop on Peer-To-Peer Systems*, February 2004.

[10] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *International Conference on Supercomputing*, June 2002.

[11] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable dynamic emulation of butterfly. In *ACM Symposium on Principles of Distributed Computing*, July 2002.

[12] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, March 2002.

[13] R. Morselli, B. Bhattacharjee, M. A. Marsh, and A. Srinivasan. Efficient lookup on unstructured topologies. *Technical Report CS-TR-4593, University of Maryland*, July 2004.

[14] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX Annual Technical Conference*, June 2004.

[15] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[16] S. Saroui, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking (MMCN)*, January 2002.

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

[18] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. In *IEEE JSAC*, January 2004.