Available online at www.sciencedirect.com
SCIENCE DIRECT
J. Parallel Distrib. Comput. 66 (2006) 257–273
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Locality in structured peer-to-peer networks

Ronaldo A. Ferreira*, Suresh Jagannathan, Ananth Grama

Department of Computer Sciences, Purdue University, 250 N. University Street, West Lafayette, IN 47907-2066, USA

Received 9 January 2004; received in revised form 23 August 2004; accepted 21 September 2005
Available online 15 November 2005

Abstract

Distributed hash tables (DHTs), used in a number of structured peer-to-peer (P2P) systems, provide efficient mechanisms for resource placement and location. A key distinguishing feature of current DHT systems, such as Chord, Pastry, CAN and Tapestry, is the way they handle locality in the underlying network. Topology-based node identifier assignment, proximity routing, and proximity neighbor selection are examples of heuristics used to minimize message delays in the underlying network. While these heuristics are sometimes effective, they all rely on a single global overlay that may install the key of a popular object at a node far from most of the nodes accessing it. Furthermore, a response to a lookup message does not contain any locality information about the nodes holding a copy of the object. We address these issues in Plethora, a novel two-level overlay P2P network. A local overlay in Plethora acts as a locality-aware cache for the global overlay, grouping nodes close together in the underlying network. Local overlays are constructed by exploiting the organization of the Internet into autonomous systems (ASs). We present a detailed experimental study that demonstrates performance gains in response time of up to 60% compared to a single global Pastry overlay. We also present efficient distributed algorithms for maintaining local overlays in the presence of node arrivals and departures.

Keywords: Peer-to-peer networks; DHT; Query resolution; Locality; Caching; Internet

1. Introduction

The past few years have seen considerable activity in the area of peer-to-peer (P2P) systems and applications. File-sharing systems, such as Napster [19] and Gnutella [12], have gained immense popularity and attracted significant research attention. As has been well-documented [6,29,28,14], these early systems had major scalability issues—Napster for using a central server for queries, and Gnutella for using an inefficient search protocol that floods the entire network, or at least a significant part of it, in order to find an item of interest. The scale of P2P networks with their large number of participating nodes requires these networks to be highly scalable in terms of aggregate resource requirement as well as end user performance. The latter requirement translates to minimizing the number of hops a message must travel in order to satisfy a query.

A number of researchers have addressed the problem of scalability in P2P networks [32,25,36,17]. Structured P2P systems such as Chord [32], Pastry [25], and Tapestry [36] provide a simple primitive for name resolution: given a key (e.g., a file name), return the location (IP addresses) of the nodes that currently have references to corresponding data objects (the files). To support this primitive, these systems rely on a *distributed hash table* (DHT) abstraction, and typically provide an upper bound of $O(\log n)$ on overlay hop-count, where n is the total number of nodes in the network. This upper bound is achieved using a small amount ($O(\log n)$) of routing information per node. Other systems such as CAN [22] support similar primitives, but have different upper bounds on hop-count subject to varying constraints on per-node routing information.

While recent work has focused on minimizing the number of overlay hops, the delay experienced by messages in the underlying network can also be a major performance bottleneck. Since nodes and objects are identified by random strings, lookup messages may travel around the world before reaching a destination node that is in the same local network as the source node. To minimize the effects of randomization, several heuristics have been incorporated into these systems. These include *proximity routing*, *topology-based node ID assignment*, and *proximity neighbor selection* [24].

* Corresponding author.
E-mail addresses: rf@cs.purdue.edu (R.A. Ferreira), suresh@cs.purdue.edu (S. Jagannathan), ayg@cs.purdue.edu (A. Grama).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2005.09.002

While these heuristics produce good results when compared to a standard implementation, they nevertheless rely exclusively on a single global overlay. Consequently, no guarantees can be provided that a popular object has its key installed close to the majority of nodes that access it. Moreover, a response to a lookup message does not contain any locality information about the nodes holding a copy of the object. Thus, a node receiving a query response has no information about which nodes are close or distant from it. In this paper, we present the routing core of Plethora, a wide-area read–write distributed file system currently under development at Purdue University [8]. In Plethora, peers are expected to be semi-static and to have good Internet connectivity in terms of bandwidth. The Plethora routing core is a two-level overlay: a global overlay, which shares the same properties as other structured P2P systems; and local overlays, which serve as locality-aware caches for the global overlay. The global overlay provides an information repository for operation and maintenance of local overlays, directing nodes to local overlays to which they belong. This two-level architecture has the advantage of localizing queries for cached objects, reducing response time, and lookup message traffic in the underlying network. Moreover, responses to objects cached in the local overlay contain pointers to nodes that are close to the node issuing the query, thus reducing congestion on the network.

The Plethora routing core's two-tier organization is motivated by studies which show that queries for multiple keys in P2P networks follow a Zipf-like distribution [31]—a small number of objects are very popular and are the targets of the majority of the queries. This difference in access frequencies suggests that a well designed caching mechanism can significantly improve the response time of the system. Another motivating factor is provided by new applications being deployed on P2P networks, for example distributed file systems like Oceanstore [15]. To provide high levels of availability, in P2P distributed file systems, files are typically partitioned (erasure encoded) into blocks and the blocks are spread over the network. If the blocks are stored randomly in the network, without any notion of locality, the retrieval of data becomes much more expensive, since several network accesses, possibly to different nodes, must be performed.

To organize nodes into local overlays, we rely on the organization of the Internet into a collection of autonomous systems (ASs). An AS is a group of networks controlled by a single administrative authority, and in most cases restricted to a geographic region (with noted exceptions). An AS is identified by a unique 16-bit identifier. When a new node joins the network, it first joins the global overlay and then uses its AS number to find other nodes of its AS that are already members of a local overlay. Each AS has a node in the global overlay, its rendezvous point [5], which is responsible for maintaining information about nodes in the AS that are present in the network.

In an ideal scenario, a local overlay contains nodes belonging to a single AS. Since the number of nodes of a particular AS participating in the overlay network may be small, we present an efficient distributed algorithm for merging local overlays. Two ASs are merged into the same local overlay if the distances among their nodes are smaller than a prescribed threshold D. By allowing nodes in multiple ASs to form a local overlay, the arrival of new nodes may increase the size of a local overlay beyond a point where performance improvements are not significant. To avoid this situation, we also present a distributed algorithm for splitting local overlays. The algorithm for splitting a local overlay guarantees that nodes in the same AS stay in the same local overlay after a split operation with high probability.

Routing messages in a local overlay uses an algorithm similar to Pastry, with nodes storing information about $O(\log\ m)$ other nodes, where m is the number of nodes in the local overlay. The number of hops is, similarly, logarithmic in the number of local nodes. This allows local overlays to be arbitrarily large. We therefore also consider the problem of ASs that span a geographical area that is too large to be considered a local overlay. It is important to note that the state information per node in our two-level architecture is just a constant factor larger than the information stored by the current systems; in other words, each node still stores $O(\log n)$ routing information.

Two parameters that impact the performance of our architecture are (i) the accuracy of topological information, and (ii) the optimality of sizes of the local overlays. There have been a number of other efforts aimed at accurately mapping the Internet topology [10,20]. The results of these efforts can be used to derive topological information. Due to its reliance on the existing Internet infrastructure using ASs, our scheme is largely resilient to inaccuracies in topological information. With respect to determining optimal sizes for local overlays, the problem can be transformed into one of partitioning a weighted graph, which is known to be NP-hard [11]. A number of heuristics (multilevel and spectral methods) as well as distributed algorithms exist for this problem. We demonstrate here that a weak heuristic based on greedy node aggregation for merging ASs is sufficient for significant performance improvements. Our main goal is to reduce the lookup latencies, we do not try to find the node with the best connection that yields the best transfer time. However, by answering a query with nodes in the same AS, we expect that our approach implicitly returns such nodes in most of the cases.

1.1. Contributions

The main contributions of this paper are summarized below:

- We present a novel caching scheme that uses Internet topology information to group nodes that are physically proximate into local overlays. Our caching scheme can be used in conjunction with any DHT system, and does not rely on the deployment of well positioned landmarks [23].
- We present an efficient distributed algorithm for splitting local overlays that preserves locality in the two new local overlays created after a split operation. We present an analysis of the probabilistic properties of this algorithm.
- To ensure that local overlays do not become very small, we also develop an efficient distributed algorithm for merging local overlays.
- We present a detailed simulation study that demonstrates the effectiveness of our caching scheme. In our simulation,

we use different workloads, both synthetic and from real file traces, to characterize object sizes stored in the network. To generate the underlying topology, we use real Internet data collected by the Skitter project [30]. The simulation study shows that our scheme can reduce response delays of the system by up to 50%, and can reduce the number of messages transmitted in the underlying network by up to 60% for different workloads and access patterns compared to a corresponding Pastry network using the proximity neighbor selection heuristic. We also show that the overheads associated with splitting and merging local overlays is very low.

The rest of the paper is organized as follows. Section 2 presents basic definitions and background information necessary for the description of our algorithms. Section 3 presents Plethora's routing core architecture and algorithms for merging and splitting local overlays. In Section 4, we present simulation results. Related work is presented in Section 5 and conclusions are drawn in Section 6.

2. Background

Current routing schemes in P2P networks, such as Chord [32], Pastry [25], and Tapestry [36], work by correcting a certain number of bits at each routing step. These schemes can be viewed as variants of hypercube (or dimension ordered) routing [16]. The routing scheme of Plaxton [21] for accessing nearby copies of objects in distributed systems is also related. In these systems, nodes and objects share the same address space. Nodes have their addresses assigned randomly and uniformly. This is generally achieved by computing a hash function on their IP addresses. Objects are identified by computing a hash function on their names. The uniform distribution of the identifiers (nodes and objects) is desirable in order to provide load balance in the system; that is, all nodes are expected to store roughly the same number of object keys. For the sake of brevity, we restrict our discussion here to Pastry and refer interested readers to the bibliography for other protocols.

In Pastry, objects and nodes are assigned random and uniformly distributed identifiers of length 128 bits. An object is stored in the node that is numerically closest to the object's identifier. Each node maintains routing information (overlay IDs and IP addresses) about a limited number of neighbors, with the size of the routing table varying depending on a configuration parameter, b, which indicates how many bits are resolved at each routing step. To route a message, each intermediate node, along the message path, tries to forward the message to a node whose node ID shares a prefix with the destination identifier that is at least b bits longer than the current node's shared prefix. If the routing table does not have any entries for such a node, the message is forwarded to a node that shares a prefix of the same length as the prefix shared by the current node, but is numerically closer to the destination. In addition to the routing table, each node maintains two other pieces of information: a *leaf set* and a *neighborhood set*. The leaf set stores information about l nodes that are numerically closest to the current node, with $l/2$ nodes having smaller IDs than the current node, and $l/2$ having larger IDs. This information is stored to provide reliable message routing, and is normally used when the routing table does not have an entry for a node that shares a prefix longer than the current node. The neighborhood set stores information about m nodes that are closest in the underlying network to the current node. While this information is not normally used for routing, it is useful in applications that exploit network locality properties. Fig. 1 illustrates an example of the state stored in a particular Pastry node.

Pastry State Information			
Leaf Set			
Smaller		Greater	
31001	31021	32021	32031
31200	32101	32101	32110
Routing Table			
0α	1α	2α	–
30α	31α	–	33α
–	321α	322α	323α
3200α	–	3202α	3203α
		–	
Neighborhood Set			
10223	03111	11330	23111
31001	32110	21333	22203

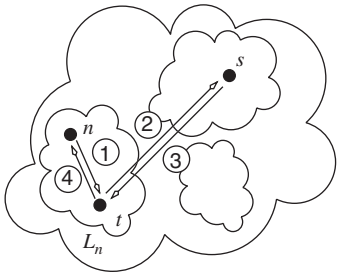
Fig. 1. Pastry state information stored at node with identifier 32012, and parameter b equal to 2. The digits are in base 4, and α is an arbitrary suffix.

An arriving node needs to initialize its tables (routing table and leaf set) and inform the other nodes of its arrival. A new node initializes its state tables by contacting a node already present in the network, and asking it to route a join message to its node ID. The nodes along the path from the contact node to the node responsible for the new node's ID send their state information to the new node. The new node uses this information and may contact additional nodes to complete its state tables.

The departure of a node x, either by voluntary termination or by failure, triggers an update in its neighbors. The nodes that are close in the ID space to the departing node must update their leaf sets. If x appears in the leaf set of a node y, y can replace x by asking the leaf set of the node with highest index on the same side of y's leaf set that x was present. If x appears in the routing table of a node w, this will be detected when w tries to use that entry. When detecting the failure of x, node w can use its leaf set to forward the message. To repair its routing table, node w can contact any node in the same row of the missing entry, and ask for its routing table. For a more detailed explanation of Pastry and its algorithms, we refer the reader to [25].

3. Plethora routing core

The Plethora routing core relies on a two-level overlay architecture [9,35,1,34]. A global overlay serves as the main data repository, and several local overlays serve as caches to improve access time to data items. When a node n needs a data item, it first searches its local overlay. If the data item is cached



① Node n sends a message to $t = h'(d)$ requesting data item d in local overlay.

② If node t has a copy of data item d, steps 2 and 3 are skipped. In step 2, node t sends a request to node $s = h(d)$ in global overlay requesting data item d.

③ Node s sends a copy of data item d to node d.

④ Node t caches a copy of data item d in local overlay L_n and forwards a copy to node n.

Fig. 2. Data access in Plethora.

in its local overlay, a copy is immediately returned. Otherwise, the data item is retrieved from the global overlay and automatically cached in n's local overlay. Fig. 2 illustrates how data items are accessed in a two-level overlay.

Local overlays contain nodes that are close to each other in the Internet. To group nodes into local overlays, we rely on the organization of the Internet into a collection of ASs. More specifically, nodes that are in the same AS should be in the same local overlay together with nodes of neighboring ASs. The global overlay can be implemented using any prefix-based DHT system. It is used as the main repository and helps direct nodes to local overlays where they belong. Since we have described Pastry in Section 2 and we adopt some of its underlying algorithms to support the local overlay, we shall use it to define the global overlay as well. We introduce one small modification: in addition to the IP addresses of neighboring nodes, we also store their AS numbers as part of the state information stored at each node. A node can determine its AS number from its IP address using a number of possible alternatives. Whois servers are the simplest, however, since the data stored in the servers is not kept up-to-date, more accurate tools can be used. Mao et al. [18] presents an accurate AS-level traceroute tool for mapping IP addresses to AS numbers. It is important to note that even if a whois server is used, the server does not constitute a bottleneck in our scheme, since its content can be conveniently replicated at peers. The server is accessed when a new node first joins the overlay network, and the new node's AS number can be locally cached for future reference.

A node must build its Internet neighborhood information over time, that is, a list of ASs that can be present in its local overlay. An initial list can be built from the routing information that the node receives when joining the global overlay. The node can measure the delay to each member of its routing table and leaf set. If the delay is less than a system parameter D, it includes the AS of the probed node in its neighborhood list. This is a nonintrusive way of acquiring neighborhood information. A more aggressive way for a node n to build the same list is by using `traceroute` to some of the nodes in n's state tables, using the IP addresses of the intermediate hops to find nearby ASs. Observe that we assume that the delay from a node in one AS to any other node in some other AS is an acceptable indication of proximity. This assumption may not be always true when we deal with ASs that span large geographical areas—we address this issue specifically in Section 3.6. To simplify the presentation of our algorithms, we initially assume that this assertion is valid.

In each local overlay there is a node, the *local overlay leader*, which controls the number of nodes in the local overlay. To avoid a complicated protocol for leader selection, we assume the local overlay leader is the node with the smallest identifier in the local overlay. We discuss later in this section how a leader can be efficiently maintained in the local overlay. The local overlay leader must issue a `split` message if the number of nodes grows above the maximum allowed, and a `merge` message if the number of nodes drops below the minimum allowed. The maximum and minimum number of nodes in a local overlay are system parameters. To avoid having the leader maintain accurate information about the peers in its local overlay, thereby having to support a complicated protocol for node departures, we assume that the local overlay leader periodically circulates a message in the network to discover the current number of nodes. This is a reasonable approach, since the local overlay size has an impact only on performance and not on the correctness of the protocol. Every node in a local overlay maintains a pointer to its current leader. This pointer is used to determine if the leader has departed or failed—in which case a new leader is selected.

Even though the maximum number of nodes in a local overlay is governed by a system parameter, we expect that this value can be arbitrarily large. We therefore require efficient algorithms to route messages within a local overlay. A simplistic scheme, such as one that stores complete information at each node, would not scale to a significant number of nodes. We use a modified version of Pastry to achieve scalability and to provide guarantees on the number of hops a message travels in the local overlay.

In the local overlay, nodes are also identified by random strings. One key difference in the identification of local overlay nodes is that the number of bits in a node's identifier can change over time. The reason for this will become clear when we present algorithms for merging and splitting local overlays. When a new node decides to join an existing local overlay, it must be informed of the current identifier length by the contact node. It then generates a random identifier of the informed length locally before asking the contact node to introduce it into the local overlay. A new node is introduced into a local overlay in the same way as in Pastry: a join message is sent by the contact node and routed to the new node identifier. Nodes along the path send their state information to the new node.

Since object identifiers are assigned random strings of fixed length in the global overlay, 128 bits in the case of Pastry, and we want the objects to have the same identifiers in both overlays, we only use the w least significant bits of the object

identifiers to decide which node will store a particular object's key; w being the number of bits currently used to identify nodes in the local overlay. The node with the identifier numerically closest to the restricted object identifier is the one responsible for storing the object's key.

3.1. Routing in the local overlay

Routing in the local overlay proceeds as in the global overlay, with one significant exception. Instead of resolving multiple bits at each hop during the routing process, we resolve only one bit per hop. This is equivalent to having $b = 1$ in Pastry. The routing table of a local overlay node, therefore, has only one column. Row i of n's routing table has a pointer to a node whose identifier shares i bits with n's identifier and differs at bit $i + 1$, with i varying from 0 to $w - 1$, where w is the current identifier length. The reason we have $b = 1$ in the local overlay is to simplify the operations of merging and splitting as discussed in Sections 3.4 and 3.5. This is not a fundamental constraint—by suitably modifying merge and split operations to multiway split and merge operations, higher values of b can be used. We also change the size of the leaf set to $2 \times k \log M$, where k is a constant greater than one, and M is the maximum number of nodes in a local overlay. This latter choice is discussed in Section 3.4. We assume nodes in the local overlays implement the proximity neighbor selection heuristic to select their neighbors.

3.2. Node arrivals

When a new node arrives, it needs to initialize its tables and also inform other nodes of its arrival. A new node first joins the global overlay, and initializes all its global overlay tables the same way as in Pastry. The AS number of the nodes is the only additional information introduced. Upon joining the global overlay, the new node must join a local overlay. To find which local overlay it should join, it computes a hash function on its AS number, and uses the resulting value to find a node in the global overlay that is responsible for keeping information about that particular AS (the *rendezvous point* of the AS).

Two situations may arise depending on the response of the rendezvous point. In the simplest case, the AS of the new node already has some nodes in the network. In this case, the rendezvous point returns a list of current nodes. The new node chooses one of the nodes as its contact in the local overlay. The contact node introduces the new node in the local overlay and sends to it a pointer to the local overlay leader. The rendezvous point also stores the new node information in its AS directory. The AS directory does not need to store all AS nodes present in the network; rather, it stores a constant number of nodes that are used as contacts. The nodes in the local overlay refresh this information periodically. We assume that the global overlay replicates the contents of its nodes appropriately. If a rendezvous point leaves, one of its neighbors becomes the new rendezvous point and has the required information of the AS.

In the second case, the new node is the first node of its AS to join the network. In this case, the new node stores information at its AS's rendezvous point and uses its neighborhood information to find a contact node in a local overlay of some other AS. If the new node can find a contact node, it executes the procedure described above to join the contact node's local overlay. If the new node cannot find any contact nodes because its neighborhood list is empty or none of the ASs in the list have nodes currently in the network, the new node starts a new local overlay. In a well populated network, the latter case is unlikely, since the neighborhood list can be built with nodes already in the global overlay.

3.3. Node departures

Nodes in the overlay networks may leave or fail without notifying their neighbors. Global overlay recovery is handled by the underlying DHT system. Node departures in Pastry are handled lazily, that is, a node departure is detected only when a node tries to contact the node that left to route a message or to access a data item. When a node detects that one of its neighbors has left, it must contact other nodes to restore its state information. The leaf set of a node n can be restored by contacting a live node y in n's leaf set. Node y sends its leaf set to n, and n fixes its information using the appropriate node. If n detects that a node x in its routing table is not responding, n can contact a live node y in n's routing table that is in the same row of x, and ask y to send n its routing table.

As in Pastry, node departures in Plethora are handled lazily. A few adjustments must be made to accommodate the differences in the routing table. Since the routing table of a local overlay node has only one column, we cannot use Pastry's implementation without modification. If n detects that a node x in its routing table is not responding, n computes the prefix that it shares with x, and appends a random string to form an identifier. A special message is routed to the computed identifier using alternate paths, via the node one row above or via the leaf set. If there is a node in the network with the same prefix shared by x and n, this node receives the routed message and replies to it with the required information.

The departure of a node also triggers updates to the state stored in the nodes other than the routing tables. When a node n detects that a node y has left the local overlay, n must notify y's rendezvous point in the global overlay of its departure. One other update that may happen is when a node n detects that the local overlay leader has left; a new local overlay leader must be chosen in this case. We use a simple approach to choose the new leader—it is the node with the smallest identifier in the local overlay. Once the leader's departure is detected, the node with the smallest identifier sends a broadcast message informing the other nodes that it is the new leader. Observe that nodes keep pointers to their successors and predecessors. Therefore, the node with the smallest ID is easily determined, it is the node that is preceded and succeeded by nodes with larger IDs.

3.4. Splitting local overlays

When the local overlay leader detects that the number of nodes currently present in the local overlay exceeds the maximum allowed, it issues a split message to the local overlay members. The split message is a simple message that does not carry any information about the nodes in the local overlay, or how the nodes will be partitioned. A node receiving a split message must decide locally how the operation will be performed. One problem that may occur during a split operation is that nodes of the same AS may end up in different local overlays. This is a situation that we would like to avoid, since local overlays may eventually degenerate and lose their primary purpose of localizing queries. Therefore, a split operation in a local overlay must preserve the following AS invariant:

> *Nodes of an autonomous system must always stay in the same local overlay after a split operation.*

The implementation of this invariant is a challenging problem in itself. We want a distributed solution where the nodes should exchange as few messages as possible to restore their state tables. To implement the split operation efficiently, we extend the information stored in the routing tables of the nodes. For each entry in a node's routing table, we add a secondary neighbor. The secondary neighbors are chosen to preserve this invariant. To implement the invariant, we use a hash function $\mathcal{H}$ that maps an AS number to the set $\{0, 1\}$. This set identifies the two overlays that are created after a split operation. A node n applies $\mathcal{H}$ to its AS number and determines the local overlay it belongs to after a split operation. The secondary neighbors in n's routing table are nodes whose hash function $\mathcal{H}$ maps to the same value as node n's. Node n can determine locally if a given node qualifies as a secondary neighbor, since nodes exchange their AS numbers along with their IP addresses.

On receiving a split message, node n discards all pointers to nodes whose hash values differ from its hash value. This operation is performed in the routing table as well as in the leaf set. Fig. 3 illustrates this operation. Observe that for correct operation of the network, it is sufficient that the leaf set of a node n contains at least one node in each direction whose hash function maps to the same local overlay as n.

After a split operation, two overlays are created. One of the new local overlays will contain the leader of the original overlay; this leader remains the leader in the new overlay. In the other overlay, a new leader must be chosen. We use the same approach described above—the node with the smallest identifier is chosen as the new leader. This node must send a message to the nodes in its overlay informing them that it is the new leader, and collecting information about which ASs are present in the new local overlay.

A split operation may result in disconnected networks depending on the number of nodes in the leaf set. The reason why we set the leaf set size equal to $2 \times k \log M$ is to guarantee that the new local overlays will be connected with high probability (whp).[1] The following lemma provides the connectivity guarantee.

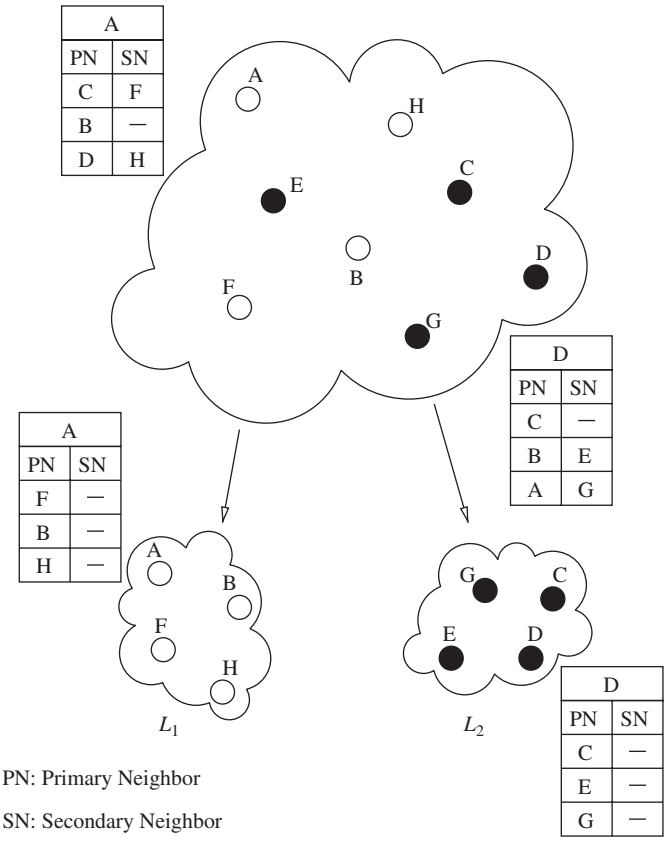


Fig. 3. Split operation. The bigger cloud contains nodes in the local overlay before the split operation. The two smaller clouds (L_1 and L_2) represent the two local overlays after the split operation. The circles of the same color represent nodes whose autonomous systems hash to the same value. The tables are the routing tables of nodes A and D before and after the split operation.

Lemma 1. *After a split operation, the two new local overlays are connected with high probability.*

Proof. Let n be a node in the original overlay and h be the hash value, computed with the function $\mathcal{H}$, of its AS number. Without loss of generality, assume that h is equal to 0. The probability of n being in a disconnected overlay after a split operation is the probability that all nodes on one side of its leaf set, assume nodes in the clockwise direction, have h equal to one. Assuming that the values 0 and 1 are equally possible, this probability is equal to

$$\left(\frac{1}{2}\right)^{k \log m} = \frac{1}{m^k},$$

where m is the number of nodes in the original local overlay. The probability of n being in a connected overlay is, therefore, $1 - \frac{1}{m^k}$. Since n was chosen arbitrarily, the result is valid for all n. □

[1] Probability equal to $1 - \frac{1}{n^{\Omega(1)}}$.

While it might appear that simply having one neighbor in the leaf set is not sufficient to guarantee connectedness, this is not the case, since we ensure that nodes are always chosen in the same (clockwise) direction. The same analysis can be used to show that, before the split operation, the routing table of a node n has secondary pointers to nodes that share the same hash value of n, whp. Consider the ith row of n's routing table, $0 \leqslant i < w$, w is the number of bits in n's identifier. Let m be the number of nodes in the local overlay, and z be the number of nodes that can be chosen for row i in n's routing table. Since we assume node identifiers are uniformly distributed, the expected value of z is equal to $\frac{m}{2^{(i+1)}}$. Since we also assume that the values 0 and 1 are equally possible as results of the function $\mathcal{H}$, the probability that row i has a secondary neighbor is equal to $1 - \left(\frac{1}{2}\right)^z$.

3.5. Merging local overlays

If the number of nodes in a local overlay L_1 drops below the minimum allowed, L_1 must be merged with a nearby local overlay L_2. If the number of nodes in L_1, m_1, is much smaller than the number of nodes in L_2, m_2, simple insertions of the nodes in L_1 are performed in L_2. However, if m_1 differs from m_2 by at most a factor α, where α is a system parameter, we use an algorithm based on the mechanism for merging two hypercubes [16]. L_1 and L_2 can be viewed as two hypercubes of dimension d that can be merged to form a hypercube of dimension $d + 1$. First, we select bit values for the overlays; for example, 0 for L_1 and 1 for L_2. The bit values chosen are sent to the nodes as part of the merge message. The nodes increase the length of their identifiers by adding their overlay bit to their identifiers as the most significant bit. A new top row is also added to the routing tables of the nodes. A node in the local overlay L_1 must fill the new row with a pointer to a node in the local overlay L_2. The same operation must be applied by the nodes in the local overlay L_2 with new pointers pointing to nodes in L_1. Since the number of nodes in L_1 and L_2 is bounded, the merge message sent in L_1 can contain the nodes in L_2, and the merge message sent in L_2 can contain the nodes in L_1. Each node chooses the new neighbor independently. Fig. 4 illustrates the merge operation. Observe that an efficient merge operation is necessary, even if the number of nodes in a local overlay is small. If a complete reorganization of the nodes were necessary in a merge operation, nodes would have to close all their current connections and open new ones.

Another way of interpreting the merge operation is the merging of two binary trees. Fig. 5 illustrates this idea. The numbers in the edges represent the identifier of the nodes at the leaves, for example, node F in L_1 has identifier equal to 01 before the merge operation and identifier equal to 001 after the merge operation. Node A's routing table has pointers to nodes F (node whose identifier differs from A's identifier in the first bit), and B (node whose identifier differs from A's identifier in the second bit, but shares the first bit). When a merge operation is executed, node A just needs to add a pointer to any

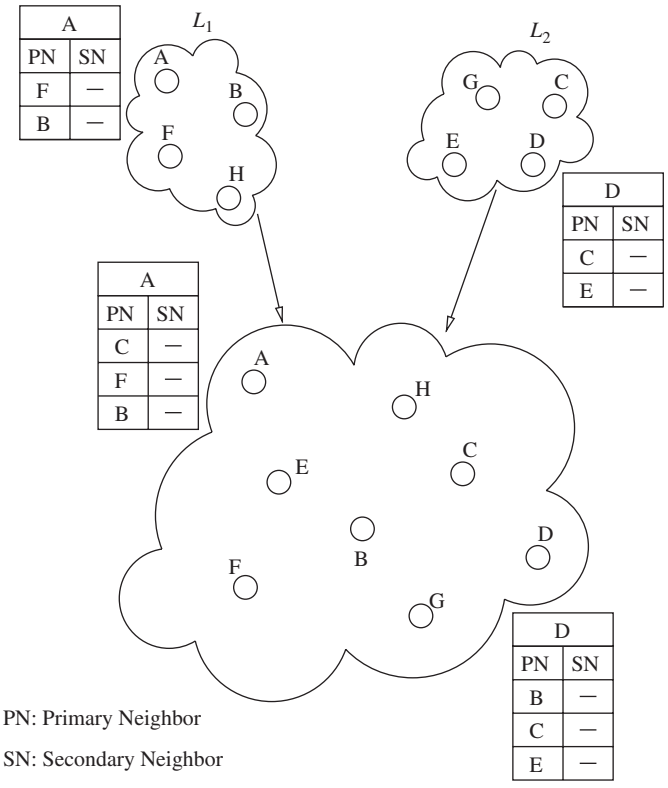


Fig. 4. Merge operation. L_1 and L_2 are the two local overlays that will be merged. The bigger cloud represents the local overlay after the merge operation. Observe that one row was added to A's routing table with a pointer to a node originally in L_2. The same thing happened in D's routing table, but here a pointer to a node originally in L_1 was added.

node on the right side of the new tree. In this example, node A adds a pointer to node C, node whose identifier (100) differ in the first bit of the new identifier of A (000). The same idea is illustrated in the routing tables of node D.

3.6. Dealing with large autonomous systems

As discussed in related work [35,34], an AS constitutes a reasonable unit of locality. Most of the current ASs in the Internet span a limited geographical area. However, there are also ASs that span a very large geographical area, such as big Internet service providers. To handle these systems, we can adapt our scheme in the following manner. When the first node n of an AS joins the network, it installs its information and declares itself a local landmark. When a node x from the same AS joins the network, it measures its delay to node n, and to other landmarks of the AS. If there is a landmark node y whose delay to node x is less than D, the maximum distance allowed, node x joins the local overlay of node y. If there is no such node y, node x declares itself a new landmark, generates a virtual AS number (larger than the maximum AS number allowed in the Internet), and either creates its own local overlay or joins the local overlay of an AS that is close to it. The virtual AS number is used, in conjunction with the IP address and the real AS number, to identify node x in the local overlay, and to

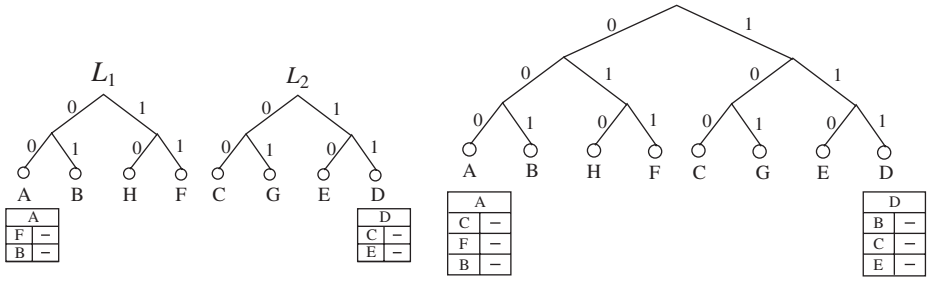


Fig. 5. Merge operation. L_1 and L_2 are the two local overlays that will be merged. The tree on the right side represents the local overlay after the merge operation.

determine the local overlay in which node x will end up when a split operation is performed.

4. Experimental results

In this section, we present simulation results for various performance aspects of the Plethora routing core. Specifically, we demonstrate considerable performance improvements (up to 50%) in response time, and show low overhead associated with control algorithms for joining, splitting, and merging local overlays. We also explore the impact of node arrivals and departures on the performance of the Plethora routing core.

Our simulation testbed implements the routing schemes of the global and local overlays, implements the algorithms for merging and splitting local overlays, and emulates the underlying network. The topology of the underlying network is generated using data from the Internet collected by the Skitter project at CAIDA (Cooperative Association for Internet Data Analysis) [30]. Skitter uses traceroute probes from several vantage points (we use data from all 24 vantage points) to collect router level topology data. The data contains link delay information, and IP addresses of the routers. We map the IP addresses to their ASs using BGP data collected by the Route Views project at University of Oregon [33]. The resulting topology contains 218,416 routers, 692,271 links, and 7704 ASs. The peers used in the simulation are attached to routers via LANs. We randomly select 10,000 routers and connect a LAN with 10 hosts to each one of them, for a total of 100,000 hosts. The delay between two nodes in the same LAN that we introduce is fixed and equal to 0.1 ms, the smallest delay found in the trace. The shortest path between two nodes in the underlying topology is computed hierarchically, like in the Internet. First, the shortest path between the two nodes' ASs is computed in the AS graph, and then the shortest paths inside the ASs are computed. The metric used in the shortest path computation inside an AS is the number of links. This computation simulates the operation of the Internet routing protocols BGP (inter-AS) and RIP (intra-AS). All experiments are performed on a 32-processor Origin 2000 with 16 GB of memory, running IRIX64.

The evaluation of our routing scheme requires appropriate dimensioning of the storage available for caching at each node, and realistic workloads. The main application target of the Plethora routing core is a distributed file system that is currently under development. Instead of focusing on a specific file system workload, for the sake of completeness, we use three different workloads for file sizes: web proxy logs, fixed-sized files, and a synthetic distribution based on recent measurement results on P2P file sizes [13]. The web proxy logs are from eight consecutive days in February 2003 collected by the NLANR project. [2] The trace contains references to 500,258 unique URLs, with a mean file size of 5144 bytes, median file size of 1663 bytes, largest file size of 15,002,466 bytes, and smallest file size of 17 bytes. The total storage requirement of the files in the trace is 2.4 Gbytes. We use the same number of files in the two other workloads. We describe the synthetic model along with its simulation results in Section 4.3.

As described in the previous section, a query that cannot be satisfied by the local overlay causes an access to the global overlay and the automatic caching of the object in the local overlay of the node issuing the query. As in PAST [26], we assume that objects are also cached in addition to keys. Due to the storage requirements of our workload, and to guarantee that the simulation would reach a steady state, the number of overlay nodes is set to 10,000. The cache size at each node varies depending on the workload. In the case of proxy logs, each node has a cache of size 5 MB (approximately 0.2% of the trace requirement), which corresponds to approximately 1000 files if we consider the mean file size. While this cache size is small considering the current storage capacity of hard disks, it is proportional to the workload used, which is composed mainly of small files. When a file is transferred over the network, it is broken in packets of size 1420 bytes, which corresponds to the payload size of TCP/IP packets transmitted in Ethernet LANs. In the case of fixed-size files, the cache size at each node is varied to accommodate between 500 to 2500 files. The impact of cache size on performance is discussed in Section 4.2.

As is well known, P2P networks are characterized by a dynamic population in which nodes join and leave the network frequently. Even though our target application is expected to have a semi-static population of nodes, the dynamic nature of the network population is believed to be an important parameter in P2P systems. Saroiu et al. [27] shows that in a Gnutella network most of the peers stay in the network for short periods

[2] http://www.ircache.nlanr.net/

Table 1
Parameters of the simulation environment

Overlay nodes	10,000
Hosts	100,000
Distinct objects	500,254
Max local overlay sizes	300; 400; 500; 1000; 2000
Max delay (D) (ms)	40; 50; 100; 200; 300
Zipf-parameter (α)	0.70; 0.75; 0.80; 0.85; 0.90

of time, while a few peers stay for long periods of time. We use the following two scenarios in our simulation:

(1) *Static*: We randomly select 10,000 peers from the 100,000 hosts in the network and these peers stay in the network throughout the simulation, with no departure or arrival of new peers.
(2) *Dynamic*: We initially select 10,000 random peers from the 100,000 hosts in the network, but have these peers leave the network. The peer session lengths are drawn from a Zipf distribution with parameter 0.7, which gives a good approximation for the results presented in [27]. We keep the number of peers constant in the simulation, when a peer leaves the network, another peer is selected among the hosts to join the network. All peers enter the network with empty caches. If a peer has been part of the network previously, its cache is emptied when it reenters the network. This is a conservative approach and we use it as a lower bound for measuring the impact of node dynamics in the network.

The main performance measurements that we investigate are the performance gains in response delay for queries, and the decrease in number of packets in the underlying network in the two-level overlay compared with a single Pastry overlay using the proximity neighbor selection heuristic. Performance gain is defined as: $g = \frac{m_1 - m_2}{m_1}$, where m_1 is the measurement (query delay or lookup messages) in Pastry, and m_2 is the measurement in the Plethora routing core. The response delay is the delay for a node to receive a response to a query, it does not include the delays to transfer the file. The source nodes of the queries are chosen randomly and uniformly, and the objects are accessed according to a Zipf-like distribution, the parameter of the Zipf distribution is varied and its impact quantified.

For the global overlay the Pastry parameters are: bits resolved per hop $b = 4$, and leaf set size $l = 32$. These parameters are also used in the single Pastry overlay. We measure the impact on performance gains of the cache hit ratio, the maximum delay parameter D used to construct local overlays, and the maximum number of nodes in a local overlay. The cache hit ratio is the fraction of all queries that are satisfied by a local overlay. We simulate both least recently used (LRU) and greedy dual size (GDS) [3] as the local cache replacement policies. The results for both policies are similar, with GDS producing results about 2% better than LRU overall; we report the results for LRU below. Various parameters of our simulation environment are summarized in Table 1.

4.1. Proxy logs

We initially discuss a series of results obtained with proxy logs as the workload, in Sections 4.2 and 4.3 we discuss specific results obtained with the two other workloads. In Section 4.1.1, we present results for a static population of peers, while the results for a dynamic population are discussed in Section 4.1.2.

4.1.1. Static population

The cache hit ratio varies depending on the α value in the Zipf distribution and the maximum number of nodes in a local overlay. Fig. 6 illustrates the cache hit ratios obtained with $\alpha = 0.70$ and $\alpha = 0.90$ and different maximum local overlay sizes. The values in the tables correspond to the minimum and maximum ratios obtained for the parameters, other values of α produce cache hit ratios in these intervals.

The number of local overlays varies depending on the maximum number of nodes allowed in a local overlay, the maximum delay (D) allowed among ASs, and the underlying topology. Fig. 7 shows the number of local overlays for the topology created from the Skitter traces.

Fig. 4.1.1 shows the performance improvements for the response delays. The values correspond to D equal to 40, 100, 200, and 300 ms. For D equal to 40 ms we observe the best results—roughly 50% improvement. Fig. 4.1.1 also shows an important result—for a large value of D the results are not only worse than the smaller values of D, but as the maximum number of nodes increases the performance also deteriorates. When D is

$\alpha = 0:70$					
Size \ Delay	40ms	50ms	100ms	200ms	300ms
300	68.23%	63.90%	65.84%	69.87%	74.15%
400	64.05%	65.33%	68.47%	72.74%	75.38%
500	65.53%	67.36%	70.15%	73.67%	76.58%
1,000	76.25%	77.42%	81.72%	84.00%	87.81%
2,000	79.31%	80.53%	84.00%	87.88%	92.23%

$\alpha = 0:90$					
Size \ Delay	40ms	50ms	100ms	200ms	300ms
300	76.34%	77.31%	74.89%	76.71%	79.76%
400	72.36%	73.38%	75.66%	78.69%	80.61%
500	73.39%	74.78%	76.83%	79.33%	81.45%
1,000	81.85%	82.79%	85.89%	87.51%	90.23%
2,000	84.04%	85.13%	87.63%	90.40%	93.61%

Fig. 6. Cache hit ratios for $\alpha = 0.70$ and $\alpha = 0.90$.

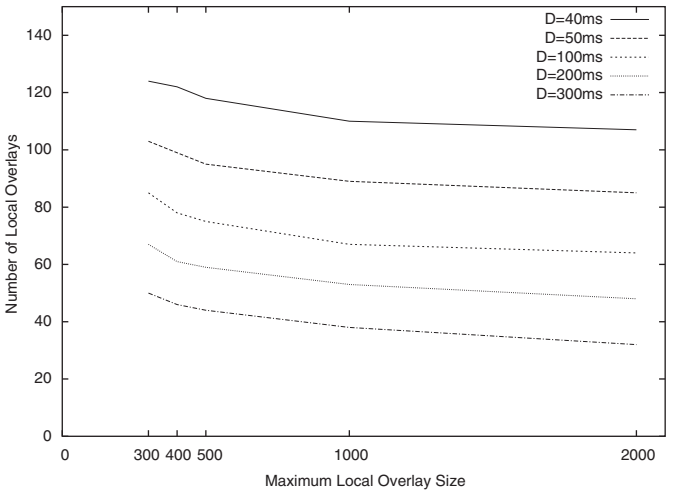


Fig. 7. Number of local overlays.

equal to 300 ms the local overlays can include nodes spread in ASs that are far apart. This is clearly not a desirable situation, since the local overlays just add additional overhead and do not serve their intended purpose of localizing traffic.

We also quantify the number of packets sent in the underlying network when we have a single overlay and compare it to the two-level overlay. Fig. 9 shows the results for D equal to 40, 100, 200, and 300 ms. The performance improvements in all configurations are in the range 30.8% to 50%, i.e., up to 50% fewer packets are sent in the underlying network.

The two-level overlay architecture of the Plethora routing core is useful only if the additional overhead of maintaining local overlays is sufficiently small. In Figs. 10 and 11 we show the results of experiments that quantify this overhead. In the first experiment, we quantify the overhead of the split operation. In this case, all 10,000 nodes join the global overlay and the local overlays simultaneously. As new nodes are added to the local overlays, split operations are required. We measure the total number of messages exchanged by nodes in the global overlay and messages exchanged by nodes in the local overlays. The overhead associated with the split operation is very small—always less than 5% for all parameters used.

To quantify the merge operation, we randomly select nodes to leave the network. As the number of nodes drops below the minimum threshold (in these experiments the minimum value is set to 20%, 30%, 40%, 50%, and 60% of the maximum value), merge operations are performed. Fig. 11 shows the result

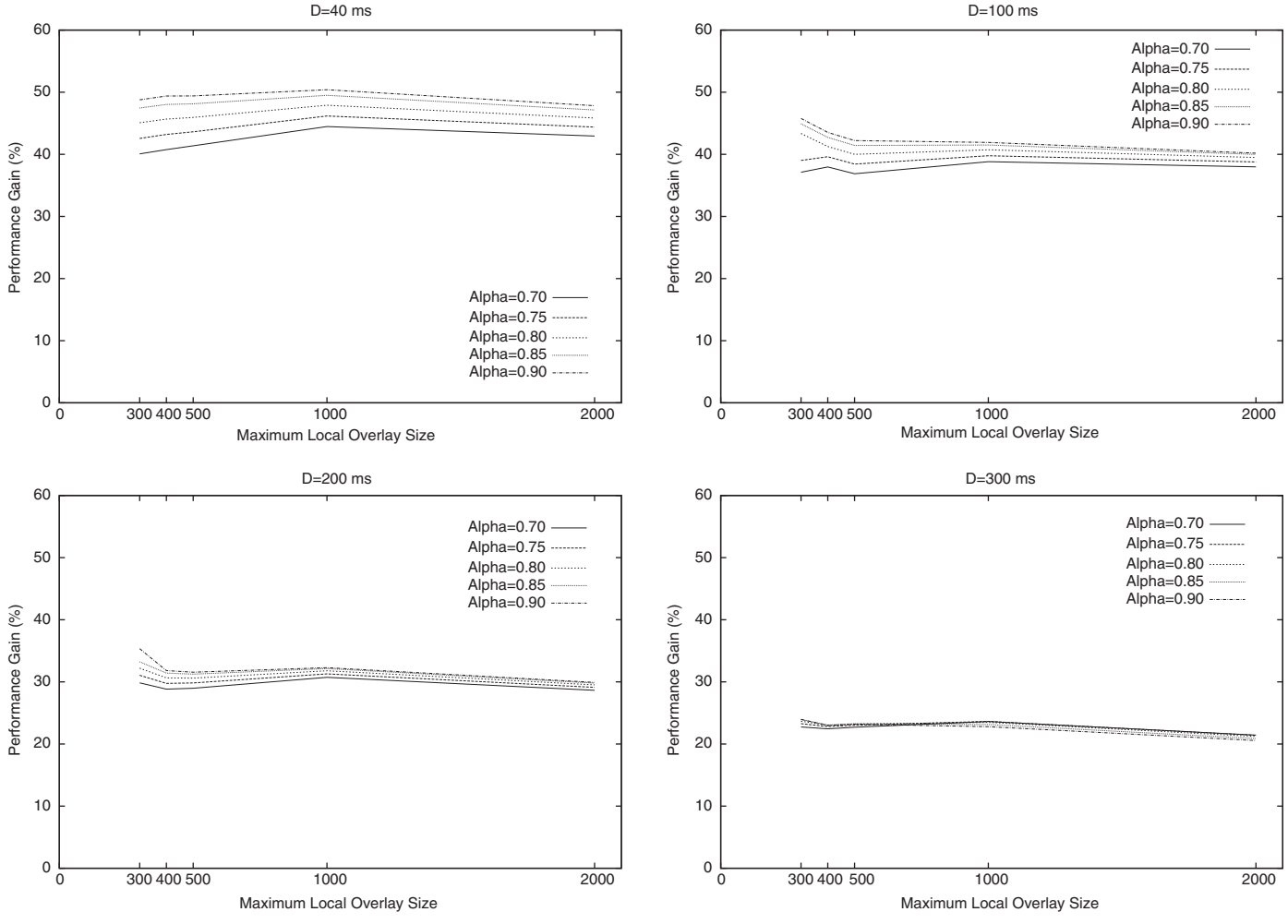


Fig. 8. Performance gains in response delay for D equal to 40, 100, 200, and 300 ms in a static population of peers.

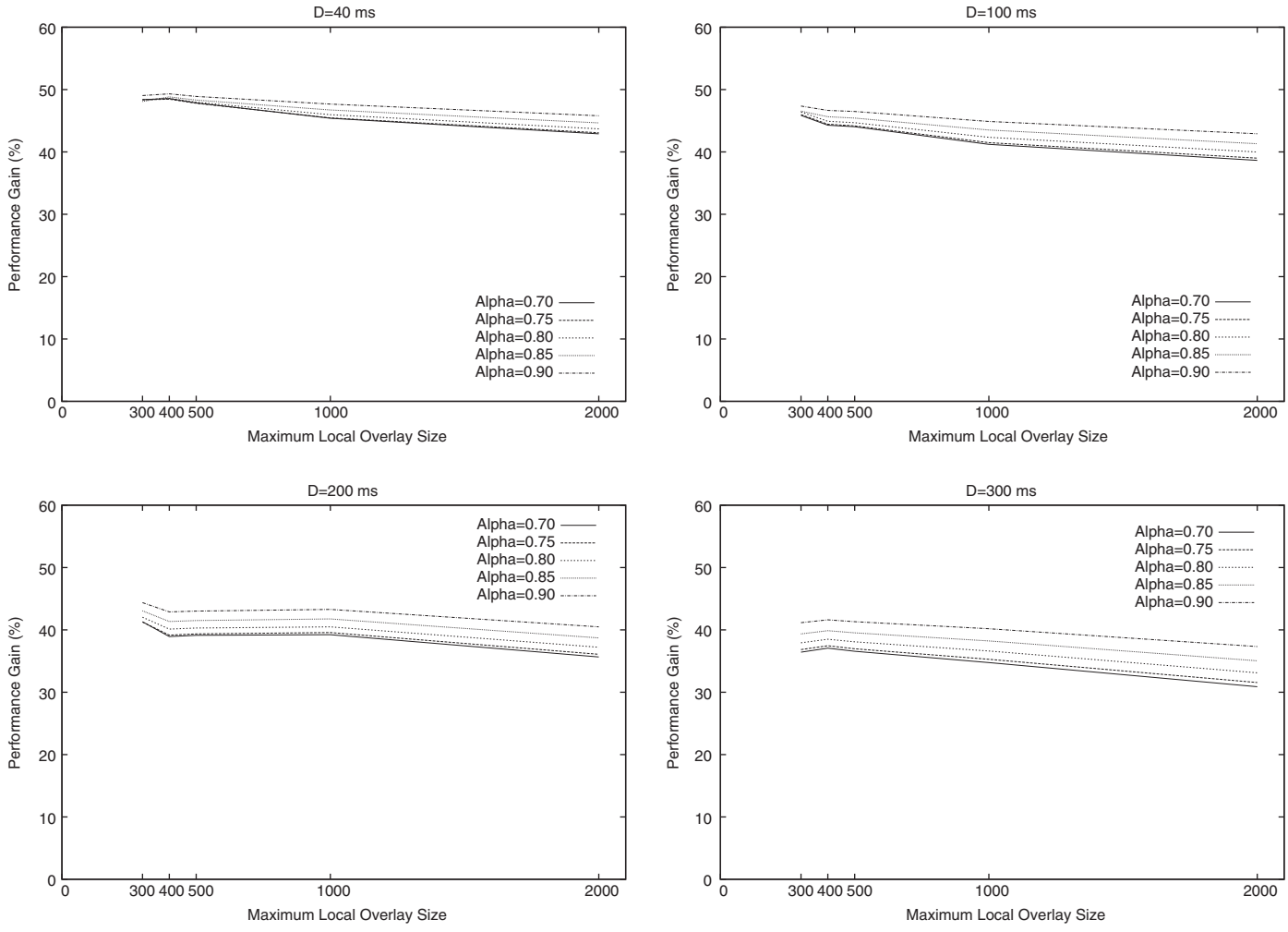


Fig. 9. Performance gains in the number of packets in the underlying network for D equal to 40, 100, 200, and 300 ms in a static population of peers.

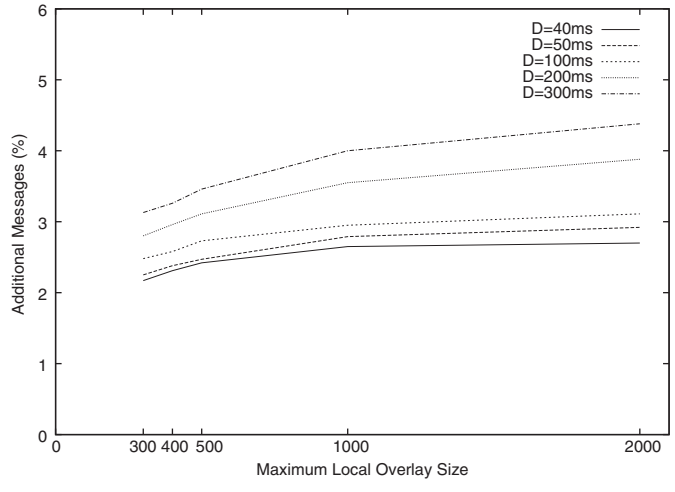


Fig. 10. Additional messages sent in the two-level overlay during the formation of the network, including messages exchanged in the split operations, when compared with a single overlay.

for 20% and 60% of the maximum size. As we can observe from the figure the merge operation is more expensive than the split operation. However, in all cases the additional overheads are smaller than 20%, with the highest value happening for a minimum size close (60%) to the maximum. The reason for this worst case is that with a threshold of 60% of the maximum local overlay size, merge operations are executed very frequently when nodes depart. Note that these fractional increases are with respect to the number of control messages in the corresponding Pastry network. In stable operation, the number of control messages can be expected to be dominated by data messages. Consequently, we expect the added overhead of our two-level overlay to be negligible.

4.1.2. Dynamic population

The next parameter we investigate is the impact of node arrivals and departures in the overlay network. Fig. 12 shows the time spent in the network by peers that joined the network during the simulation. In this particular example, 47,520 different peers participate in the network. More than 50% of the peers stay in the network for less than 10% of the total simulated time, only 5% of the peers stay for more than 60% of the total simulated time, and only 0.5% stay for more than 80%. No peer stays in the network for more than 84% of the simulated time.

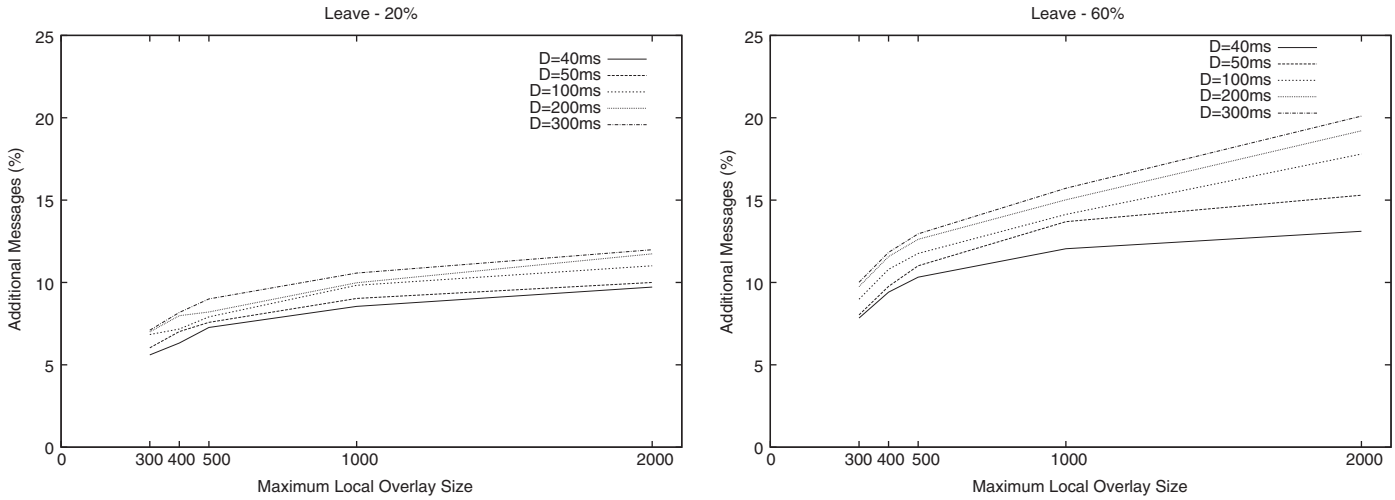


Fig. 11. Additional messages sent in the two-level overlay when nodes leave the network, including messages exchanged in the merge operations. Left: minimum local overlay size equal to 20% of the maximum local overlay size. Right: minimum local overlay size equal to 60% of the maximum local overlay size.

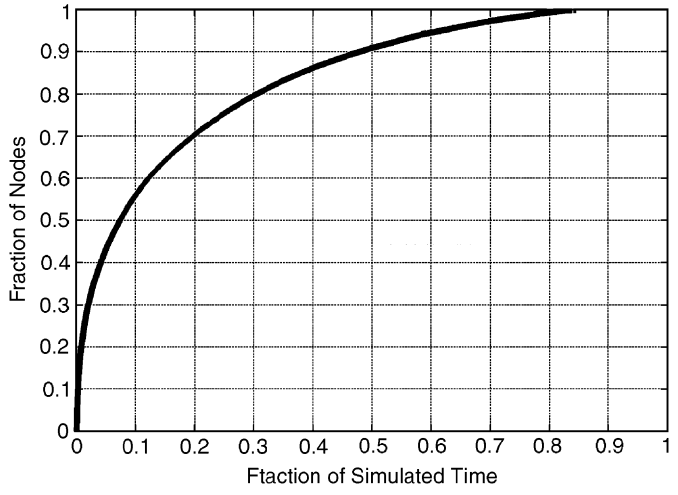


Fig. 12. Time peers stay in the network.

Figs. 13 and 14 show the performance gains in response delay and in the number of packets in the underlying network when peers are allowed to leave the network and new peers join. As we can see in the figures, the performance decreases when compared with a static population of peers, as expected. Even though there is a decrease in performance, the two-level overlay architecture can still yield gains of up to 40% depending on the configuration parameters. In a dynamic scenario, the parameter D (maximum delay between ASs) becomes more critical than in a static scenario. As we can see in the figures, for large values of D the performance gains drop to under 20%.

4.2. Fixed-size file workload

We now evaluate the impact of different cache sizes on performance gains using fixed-size files. Even though fixed-size files are not expected to be found in real networks, we use this workload for two reasons: first, we want to measure the impact of cache sizes on performance in a more controlled environment than the proxy logs would allow, that is, the maximum number of files per node is exactly the same for all peers independent of access patterns; second, a number of P2P applications, like Oceanstore, use erasure-encoded data stored in fixed-size blocks. In the experiments described below, file sizes are assumed to be one packet unit. When a file is transferred only one packet is generated in the underlying network. Cache sizes follow the same size unit.

Fig. 15 shows the performance gains in response delay and number of packets for different cache sizes. The objects are accessed using a Zipf distribution with parameter 0.9. As we can see, the performance gains increase significantly when the cache size is increased from 200 to 500 units, but beyond this point, the increase in performance is not as significant. This is expected, since small caches (200 and 500) are enough to store the most popular objects. As the cache size increases beyond these values, the additional cache space is used to store objects that are not accessed frequently.

The performance gains in number of packets (right figure) is much lower than the ones observed with proxy logs. The reason is that all objects have unit size, and, therefore, use a single packet in the underlying network. This result can be interpreted as a saving in the number of links that are used in the transfers. As the file sizes increase, as is the case for proxy logs, the observed gains also increase. We will revisit this point in the next section when we perform experiments with large files.

4.3. Synthetic workload

Gummadi et al. [13] presents an extensive study on measurements from a P2P network. Among the findings reported is how object sizes and accesses to objects are performed in a P2P system. According to the study, the file size distribution can be divided in the following thresholds: less than 10 MB, 10–100 MB, and over 100 MB. For these three intervals, the paper

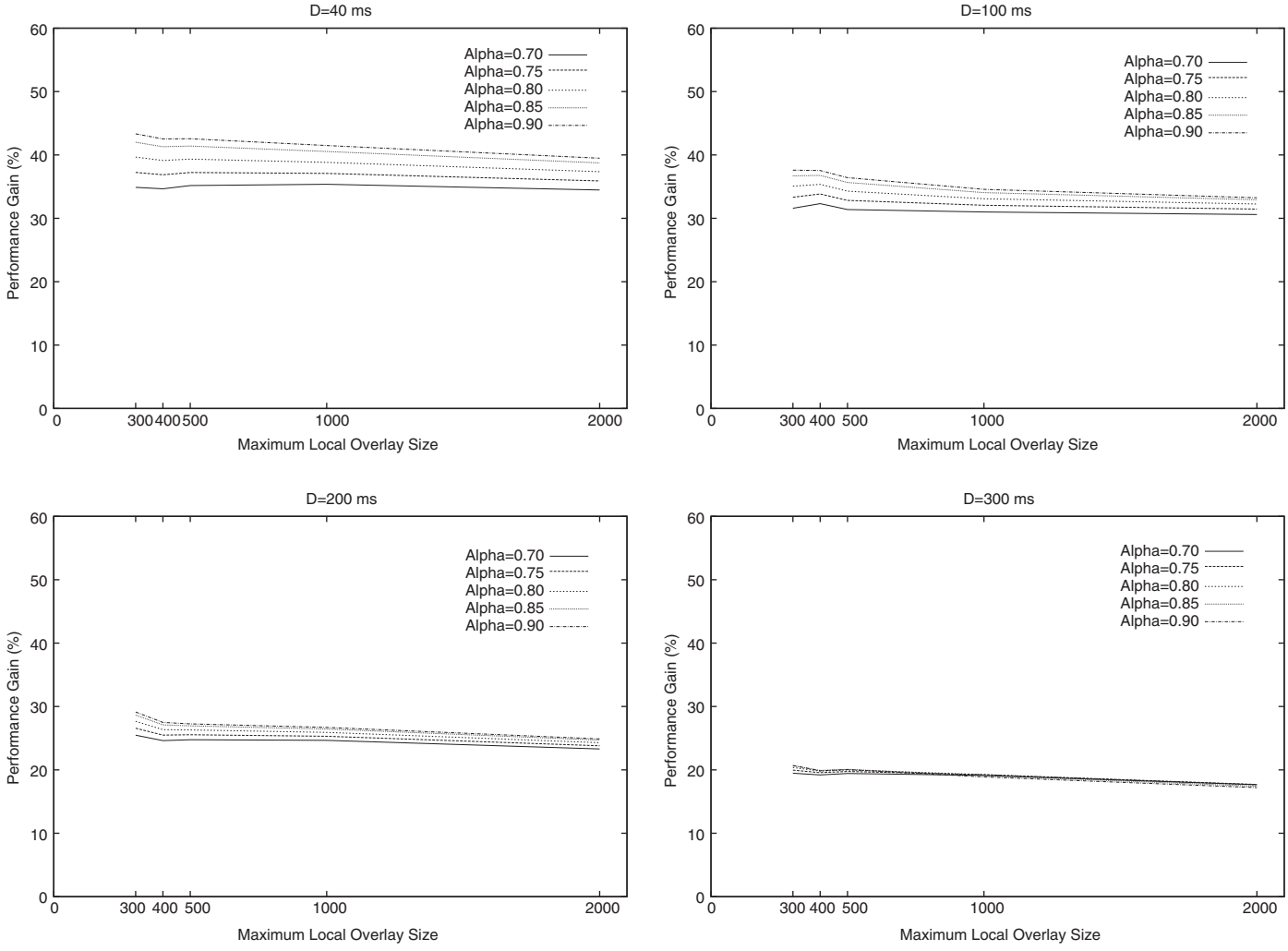


Fig. 13. Performance gains in response delay for D equal to 40, 100, 200, and 300 ms in a dynamic network.

shows that the majority of the requests (approximately 90%) are for objects smaller than 10 MB. We emulate these results by generating a synthetic distribution according to the thresholds they describe in Fig. 4 of [13]. Fig. 16 shows the cumulative distribution function of the number of requests generated as a function of object size that we use in our experiment. The object sizes vary from 100 K to 1 GB and most of the accesses are concentrated in the interval 1–10 MB. According to [13], this interval contains audio files, which correspond to the majority of files shared in current P2P networks. The mean file size in the generated trace is slightly over 75 MB and the median is 4.5 MB.

Fig. 5 shows the performance improvements when the objects and the accesses are drawn from the distribution in Fig. 16. For this experiment, we set the cache size in each node to 4.5 GB, which corresponds to roughly 1000 objects if we consider the median file size. The improvements in response delay are a little lower than the ones we observed in the experiments with the other workloads. This can be explained by the reduction in the cache hit ratio that we observe when large files are stored in the cache. When a large file is brought to the cache of a local overlay node, several small files have to be evicted from its cache. One interesting result we observe is that, even though the cache hit ratio is reduced, the improvements in number of packets in the underlying network are higher than the ones observed with the other workloads. This is mainly due to the sizes of the files, since the files are much larger, when we reduce the number of links for a transfer, the reduction in the number of packets in the network is more pronounced.

5. Related work

Three major approaches have been proposed for topology-aware overlay construction in DHT networks: *proximity routing*, *topology-based node ID assignment*, and *proximity neighbor selection* [24].

In proximity routing, each node determines the next hop for a message taking into consideration not only the node that makes more progress in the resolution of the virtual identifier, but also the proximity of its neighbors in the underlying network. This technique has been used with some success [7]. The key idea is to select, among all neighbors, the node that is closest in

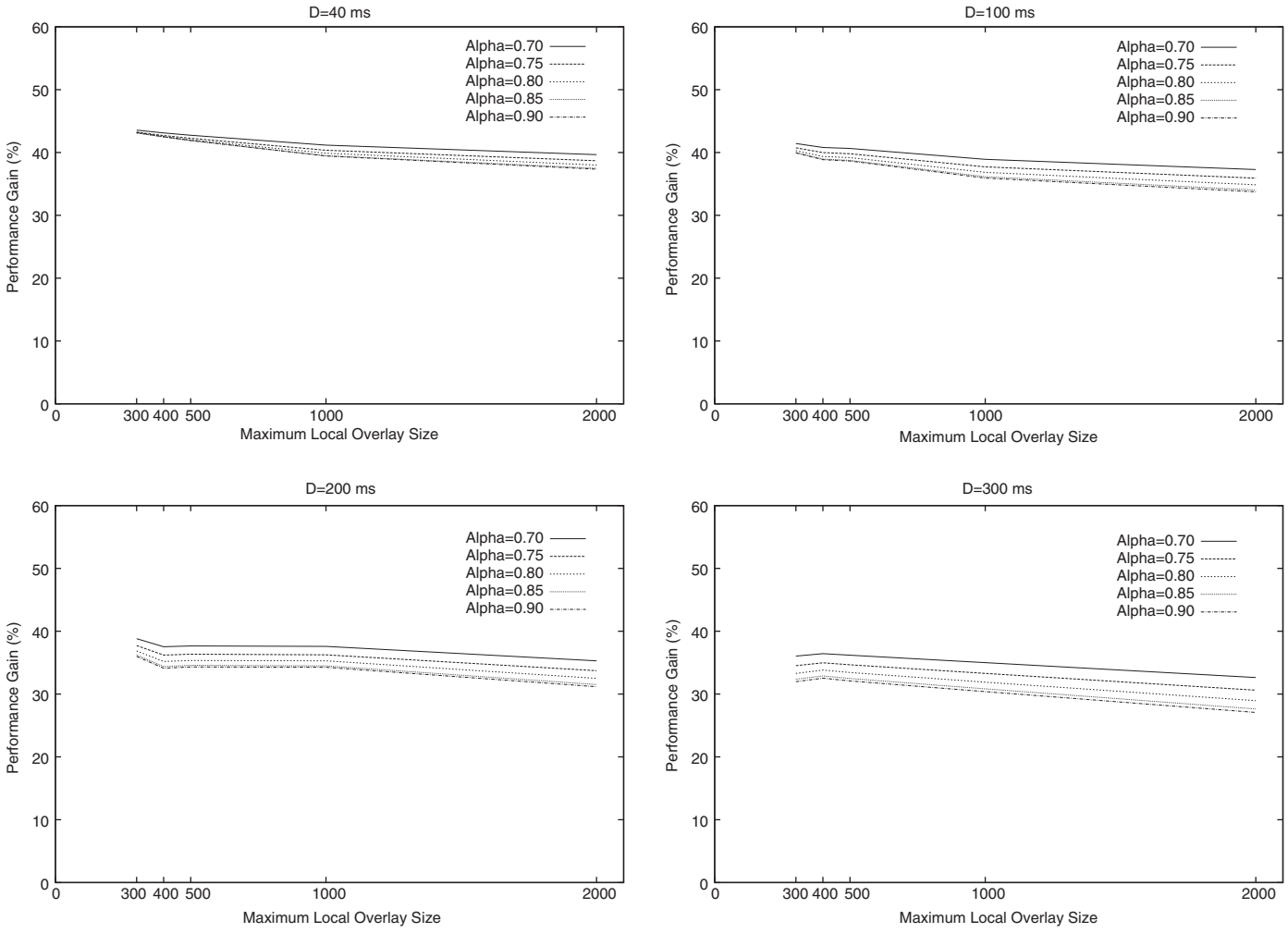


Fig. 14. Performance gains in the number of packets in the underlying network for D equal to 40, 100, 200, and 300 ms in a dynamic network.

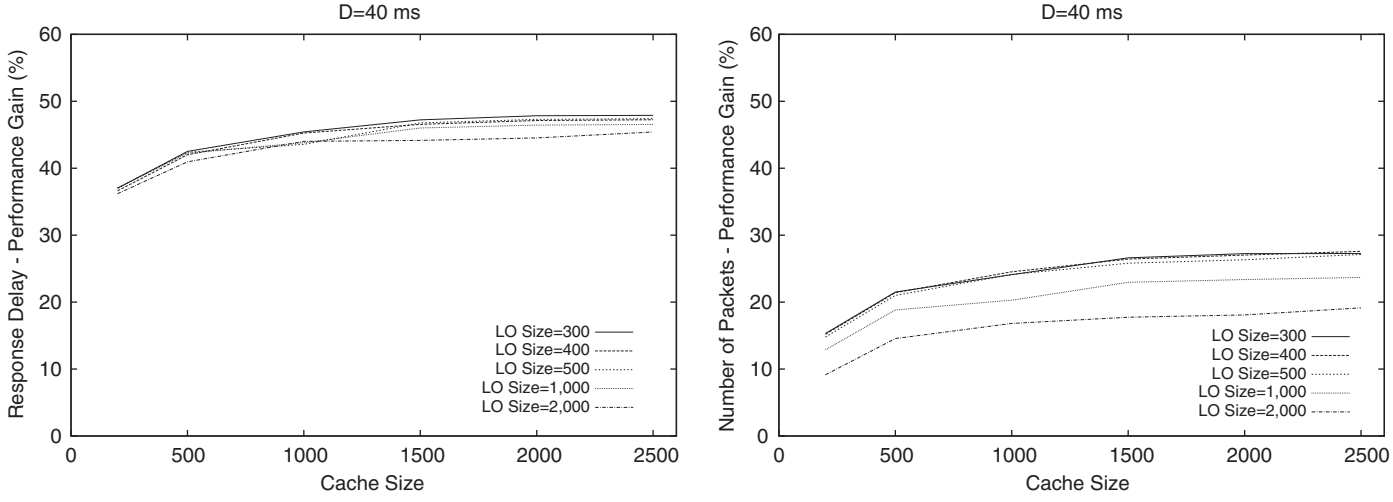


Fig. 15. Performance gains in response delay (left) and number of packets (right) for fixed-size objects and different cache sizes.

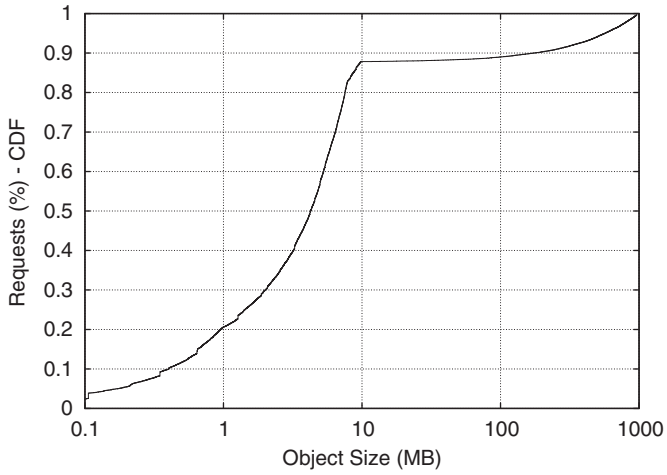


Fig. 16. Synthetic distribution of requests generated as a function of object sizes.

the underlying network or one that balances the proximity with progress in the resolution of the virtual ID. The problem with this approach is that the number of overlay hops may increase considerably.

In topology-based node ID assignment, the overlay node IDs are chosen based on the location of the nodes in the underlying network. The idea here is that nodes that are close in the underlying network are assigned overlay IDs that are numerically close. Ratnasamy et al. [23] demonstrates the use of this technique in CAN. In this particular example, before joining the network, nodes measure their distances to a set of landmarks and use the measurements to position themselves in the multi-dimensional address space of CAN. Even though this technique is able to achieve considerable reduction in routing delays, it also presents a few problems. By biasing identifier assignments, it potentially destroys the uniform distribution of nodes in the ID space.

In proximity neighbor selection, the topology of the underlying network is explored during the construction and maintenance of the routing table. For a given routing table entry, the idea is to choose, among all the nodes that qualify for that entry, the one that is closest in the underlying network to the current node. This heuristic is suitable for prefix-based protocols like Tapestry and Pastry. In these systems, the first rows of the routing table have many options for each entry, with latter rows having exponentially fewer options. As a consequence, the total delay during the routing process is dominated by the last hop. Castro et al. [4] presents a detailed study of proximity neighbor selection in Pastry. They show that the heuristic results in small relative delay penalties, without compromising the load balance of the system. Relative delay penalty is defined as the ratio of the delay experienced by a Pastry message to the delay from its source and destination nodes in the underlying network. We use this approach in the routing tables of the local overlay nodes.

The use of a two-level architecture to improve overlay performance is not new. Brocade [35] uses a secondary overlay network of supernodes. The supernodes are nodes with good capacity and network connectivity and are assumed to be close to network access points such as routers. Nodes inside an AS use the supernodes to access objects in the global overlay. Our approach differs from Brocade in several important respects. In Brocade, a normal node (not a supernode) participates in the overlay via supernodes. A normal node first needs to contact a supernode and to ask it to route its messages. Supernodes have to keep information about all overlay nodes inside their ASs. The Brocade organization is basically an overlay of servers that have several clients connected to them. There is no deterministic distributed way for a normal node to find a supernode. It is assumed that the supernodes are able to snoop the underlying network and detect overlay traffic, or that the supernodes have well known DNS names.

Xu et al. [34] also proposes a two-level overlay, consisting of one auxiliary overlay, called *expressway*, composed of

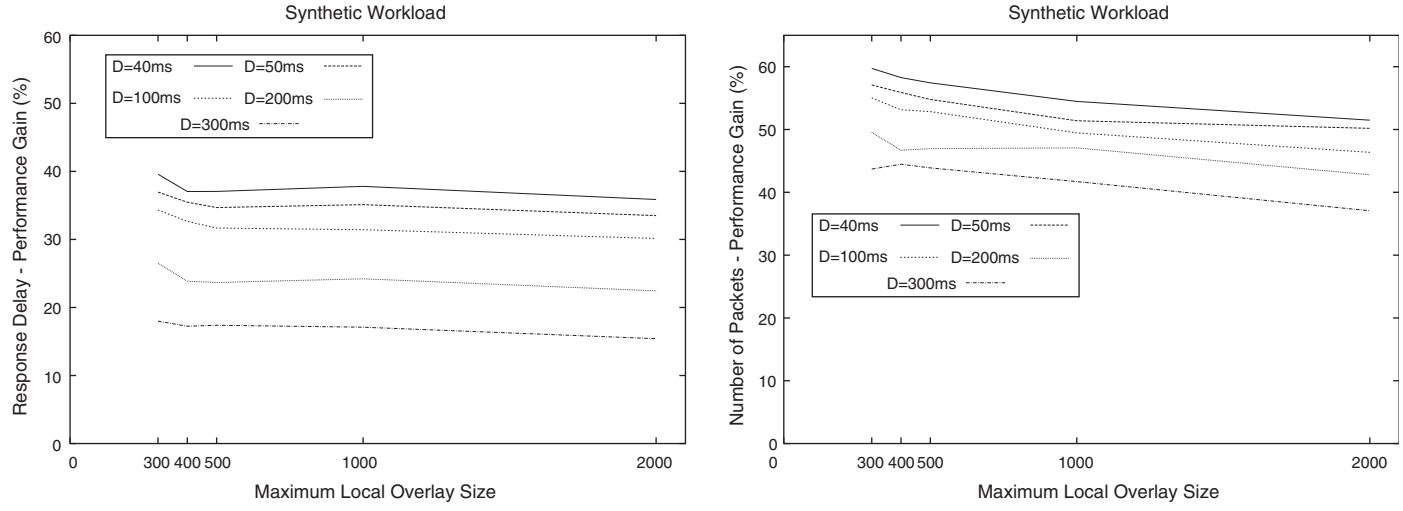


Fig. 17. Performance gains in response delay and number of packets for D equal to 40, 100, 200, and 300 ms and a synthetic workload.

supernodes, as in Brocade, and a global overlay. The global overlay is used to find nodes in the same AS or nodes that are physically close, as in our scheme. Nodes in the expressway exchange routing information, much in the same way as routers exchange BGP reports in the Internet. Our scheme resembles this approach in the way a node finds information about other nodes that are physically close to it by using the global overlay as a rendezvous point. A key difference is that the auxiliary network in [34] is intended to speed up the communication of nodes far apart in the Internet. Our auxiliary network (local overlay) is intended to speed up the communication among nodes that are close to each other, taking advantage of possible common interests, and reducing traffic in the global network. Furthermore, the expressway requires significant work to be performed by the supernodes. The supernodes try to emulate, in the overlay network, the BGP routing protocol of the Internet, by running a distance-vector routing protocol to summarize routes.

A number of measurement-based protocols [5,2], which use estimate of delay between overlay nodes, have been proposed to build overlay networks with low stretch. The main motivation of these protocols are application-level multicast. Narada [5] creates an overlay topology that minimizes the latency between nodes, but requires nodes to probe every other node. It has poor scalability, since nodes must probe all other nodes. Nice [2] addresses the scalability problem by creating a hierarchy of node clusters. Nodes are grouped in clusters based on their proximity in the underlying network. Both Narada and Nice try to build near optimal multicast trees, however, they do not provide protocols for placing and locating objects.

6. Concluding remarks

In this paper, we present Plethora, a two-level overlay architecture whose goal is to enhance locality in DHT systems. We show significant performance improvements with respect to state-of-the-art structured peer-to-peer systems. Specifically, we demonstrate performance gains of up to 60% for realistic network and workload scenarios. We also demonstrate low overheads associated with various control operations and resource requirements for Plethora.

Acknowledgments

This research has been partially funded by National Science Foundation Grants CCF-0444285, DMR-0427540, CCF-0325227, STI-5011078, and CNS-0509387.

The first author has been partially funded by CNPq and UFMS, Brazil.

References

[1] A. Agrawal, H. Casanova, Clustering hosts in P2P and global computing platforms, in: Proceedings of the third International Workshop on Global and Peer-to-Peer Computing, Tokyo, Japan, 2003, pp. 367–373.

[2] S. Banerjee, B. Bhattacharjee, C. Kommareddy, Scalable application layer multicast, in: Proceedings of the 2002 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Pittsburgh, PA, 2002, pp. 205–217.

[3] P. Cao, S. Irani, Cost-aware WWW proxy caching algorithms, in: Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS), Monterey, CA, 1997, pp. 193–206.

[4] M. Castro, P. Druschel, Y. Hu, A. Rowstron, Exploiting network proximity in peer-to-peer overlay networks, Technical Report MSR-TR-2002-82, 2002.

[5] Y. Chu, S.G. Rao, H. Zhang, A case for end system multicast, in: Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Santa Clara, CA, 2000, pp. 1–12.

[6] E. Cohen, S. Shenker, Replication strategies in unstructured peer-to-peer networks, in: Proceedings of the 2002 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Pittsburgh, PA, 2002.

[7] F. Dabek, M. Kaashoek, D. Karger, R. Morris, I. Stoica, Wide-area cooperative storage with CFS, in: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), Lake Louise, Canada, 2001, pp. 202–215.

[8] R. Ferreira, A. Grama, S. Jagannathan, Plethora: a wide-area read-write object repository, URL: http://www.cs.purdue.edu/homes/rf/plethora/.

[9] R.A. Ferreira, A. Grama, S. Jagannathan, An IP address-based caching scheme for peer-to-peer networks, in: Proceedings of the IEEE Globecom 2003, San Francisco, CA, 2003, pp. 3845–3850.

[10] P. Francis, S. Jamin, V. Paxson, L. Zhang, D.F. Gryniewicz, Y. Jin, An architecture for a global internet host distance estimation service, in: Proceedings of the IEEE INFOCOM 1999, New York, NY, 1999, pp. 210–217.

[11] M.R. Garey, D.S. Johnson, Computers and Intractability—A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, New York, 1979.

[12] Gnutella, URL: http://gnutella.wego.com/.

[13] K.P. Gummadi, R.J. Dunn, S. Saroiu, S.D. Gribble, H.M. Levy, J. Zahorjan, Measurement, modeling and analysis of a peer-to-peer file sharing workload, in: Proceedings of the 19th ACM Symposium of Operating Systems Principles (SOSP), New York, NY, USA, 2003, pp. 314–329.

[14] T. Hong, Performance, in: Peer-to-Peer: Harnessing the Power of Disruptive Technologies, O'Reilly, 2001.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, OceanStore: an architecture for global-scale persistent storage, in: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), Cambridge, MA, 2000, pp. 190–201.

[16] F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann, Los Altos, CA, 1992.

[17] D. Malkhi, M. Naor, et al., Viceroy: a scalable and dynamic emulation of the butterfly, in: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC), Monterey, CA, 2002, pp. 183–192.

[18] Z.M. Mao, J. Rexford, J. Wang, R.H. Katz, Towards an accurate AS-level traceroute tool, in: Proceedings of the 2003 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, Karlsruhe, Germany, 2003, pp. 365–378.

[19] Napster, URL: http://www.napster.com/.

[20] T.E. Ng, H. Zhang, Predicting internet network distance with coordinates-based approaches, in: Proceedings of the IEEE INFOCOM 2002, New York, NY, 2002.

[21] C.G. Plaxton, R. Rajaraman, A.W. Richa, Accessing nearby copies of replicated objects in a distributed environment, Theory Comput. Systems 32 (1999) 241–280.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content-addressable network, in: Proceedings of the 2001 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, San Diego, CA, 2001, pp. 247–254.

[23] S. Ratnasamy, M. Handley, R. Karp, S. Shenker, Topologically-aware overlay construction and server selection, in: Proceedings of the IEEE INFOCOM 2002, New York, NY, 2002.
[24] S. Ratnasamy, S. Shenker, I. Stoica, Routing algorithms for DHTs: some open questions, in: Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS'02), Boston, MA, 2002, pp. 45–52.
[25] A. Rowstron, P. Druschel, Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems, in: Proceedings of the 2001 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, San Diego, CA, 2001, pp. 247–254.
[26] A. Rowstron, P. Druschel, Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility, in: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), Lake Louise, Canada, 2001, pp. 188–201.
[27] S.D.G.S. Saroiu, K.P. Gummadi, Measuring and analyzing the characteristics of Napster and Gnutella hosts, Multimedia Systems J. 9 (2) (2003) 170–184.
[28] S. Saroiu, K.P. Gummadi, S.D. Gribble, A measurement study of peer-to-peer file sharing systems, in: Proceedings of the Multimedia Computing and Networking 2002 (MMCN '02), San Jose, CA, USA, 2002.
[29] S. Sen, J. Wang, Analyzing peer-to-peer traffic across large networks, ACM Trans. Networking 12 (2) (2004) 219–232.
[30] Skitter Project, URL: http://www.caida.org/tools/measurement/skitter/
[31] K. Sripanidkulchia, The Popularity of Gnutella Queries and its Implication on Scalability, February 2001, URL: http://www.cs.cmu.edu/kunwadee/research/p2p/gnutella.html
[32] I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup service for internet applications, in: Proceedings of the 2001 ACM SIGCOMM Conference on Applications Technologies, Architectures, and Protocols for Computer Communication, San Diego, CA, 2001, pp. 149–160.
[33] University of Oregon Route Views Project, URL: http://antc.uoregon.edu/route-views/
[34] Z. Xu, M. Mahalingam, M. Karlsson, Turning heterogeneity into an advantage in overlay routing, in: Proceedings of the IEEE INFOCOM 2003, San Francisco, CA, 2003.
[35] B. Zhao, Y. Duan, L. Huang, A. Joseph, J. Kubiatowicz, Brocade: Landmark routing on overlay networks, in: Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS), Cambridge, MA, 2002, pp. 34–44.
[36] B.Y. Zhao, J. Kubiatowicz, A.D. Joseph, Tapestry: an infrastructure for fault-tolerant wide-area location and routing, Technical Report UCB/CSD-0101141, UC Berkeley, Computer Science Division, April 2001.

Ronaldo A. Ferreira received his B.Sc. degree in computer science from the Federal University of Mato Grosso do Sul-Brazil in 1992 and his M.Sc. degree in computer science from the University of Campinas-Brazil in 1998. He worked as a lecturer at the Federal University of Mato Grosso do Sul from 1995 to 2000, when he joined the Ph.D. program in the Computer Science Department at Purdue University, where he is currently a Ph.D. candidate. His research interests are in networking and distributed systems.

Suresh Jagannathan is currently an associate professor at Purdue University. Prior to joining Purdue, he was a senior research scientist at the NEC Research Laboratories, Princeton. His interests span programming languages, concurrent and distributed systems. He holds a Ph.D. from MIT.

Ananth Grama received his Ph.D. from the University of Minnesota in 1996. He is currently an Associate Professor of Computer Sciences and a University Faculty Scholar at Purdue University. His research interests span the areas of parallel and distributed computing architectures, algorithms, and applications. On these topics, he has authored several papers and co-authored a text book 'Introduction to Parallel Computing' with Anshul Gupta, George Karypis and Vipin Kumar. He is a member of American Association for Advancement of Sciences.