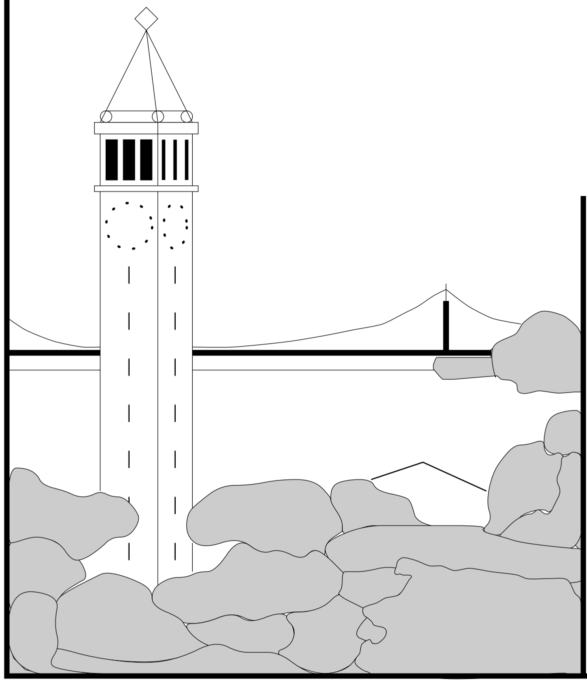
Distributed Data Location in a Dynamic Network

Kirsten Hildrum, John D. Kubiatowicz,
Satish Rao and Ben Y. Zhao
Computer Science Division
University of California at Berkeley
{`hildrum, kubitron, satishr, ravenben`}`@cs.berkeley.edu`



Report No. UCB/CSD-02-1178

April 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Distributed Data Location in a Dynamic Network

Kirsten Hildrum, John D. Kubiatowicz,
Satish Rao and Ben Y. Zhao
Computer Science Division
University of California at Berkeley
{hildrum, kubitron, satishr, ravenben}@cs.berkeley.edu

April 2002

Abstract

Modern networking applications replicate data and services widely, leading to a need for *location-independent routing* – the ability to route queries directly to objects using names that are independent of the objects' physical locations. Two important properties of a routing infrastructure are *routing locality* and *rapid adaptation* to arriving and departing nodes. We show how these two properties can be achieved with an efficient solution to the nearest-neighbor problem. We present a new distributed algorithm that can solve the nearest-neighbor problem for a restricted metric space. We describe our solution in the context of Tapestry, an overlay network infrastructure that employs techniques proposed by Plaxton, Rajaraman, and Richa [16].

1 Introduction

In today's chaotic network, data and services are mobile and replicated widely for availability, durability, and locality. This has lead to a renewed interest in techniques for routing queries to objects using names that are independent of location. The notion of *routing* is that queries are forwarded from node to node until they reach their destinations. The importance of the *location-independent routing* problem has spawned a host of proposals, many of them in the context of data sharing infrastructures such as OceanStore [13], FarSite [3], CFS [11], PAST [8]. To permit locality optimizations, it is important that the routing process use as few network hops as possible and that these hops should be as short as possible.

The set of properties that a routing infrastructure should exhibit is small but significant:

1. *Deterministic Location*: Objects should be located if they exist anywhere in the system.
2. *Routing Locality*: Routes should have low *stretch*[1], sending queries over the shortest path possible to satisfy them.
3. *Minimality and Load Balance*: The infrastructure must not place undue stress on any of its components; this implies minimal storage and balanced computational load.
4. *Dynamic Membership*: The infrastructure must adapt to arriving and departing nodes while maintaining the above properties.

Although clearly desirable, the first property is not guaranteed by most of the deployed peer-to-peer systems such as Gnutella [14] and FreeNet [5]. This paper will argue that the last three properties are closely related to one another and are achieved by an infrastructure that is capable of solving the nearest-neighbor problem efficiently.

A simple routing scheme would employ a central directory of object locations. Object servers would *publish* the existence of objects by inserting entries into the central directory. Clients would send *queries* to the directory, which forwards them to their destination. This solution, while simple, induces a heavy load on the directory server. Moreover, when a nearby server happens to contain the object, the client must still interact with the directory server which may be quite far away. The average routing latency of this technique is $O(d)$, where d is the diameter of the network – regardless of the actual distance to the object. Worse, it is fault tolerant, since the directory becomes a single point of failure for the system.

Several recent proposals, Chord [18], CAN [17] and Pastry [9], address the load aspect of this problem by distributing the directory information over a large number of nodes. In particular, they can can find an object with polylogarithmic number of application-level network hops while ensuring that no node contains much more than its share of the directory entries. Moreover, they can support the introduction and removal of new participants in the peer-to-peer network. Unfortunately, these approaches significantly increase the network latency of finding the object over even the obvious centralized directory solution.

Another solution is to publish an object's location to every node in the network. This solution allows clients to easily find the nearest copy of the object, but requires a lot of work (and network bandwidth) to publish the object. Further, this solution requires knowledge of the participants of the network. In a dynamic network, maintaining a list of participants is a problem in its own right.

In this paper, we describe the Tapestry overlay routing and location infrastructure. Tapestry uses as a starting point the distributed data structure proposed by Plaxton, Rajaraman, and Richa [16]. Henceforth, we will refer to this as the PRR scheme. This proposal provided routing locality (although with a complex routing scheme), and reasonably balanced storage and computational load. What it did not provide, however, was dynamic maintenance of membership. The original statement of the algorithm required a static set of

[1]Stretch is the ratio between the distance traveled by a query on its way to an object and the minimal distance to the object.

Scheme	Dynamic	Space	Stretch,Metric	Hops	Balanced?
CHORD [18]	$O(\log^2 n)$	$O(n \log n)$	-	$O(\log n)$	yes
CAN [17]	$O(r)$	nr	-	$rn^{1/r}$	yes
Pastry [9]	yes	$O(n \log n)$	-	$O(n \log n)$	yes
This Paper (Tapestry)	$\boldsymbol{O(\log^2 n)}$	$O(n \log n)$	-	$O(\log n)$	yes
Awerbuch, Peleg[1]	no	$O(n\delta^2 + n\delta \log^2 n)$	$O(\log^2 n)$,general	$O(\log^2 n)$	no
PRR [16]	no	$O(n \log n)$	$O(1)$,special	$O(\log n)$	yes
PRR + This Paper	$\boldsymbol{O(\log^2 n)}$	$O(n \log n)$	$O(1)$,special	$O(\log n)$	yes
PRR v.0 + This Paper	no	$O(n \log^2 n)$	$\boldsymbol{O(\log^3 n)}$**,general**	$O(\log^2 n)$	no

Table 1: In this table, n is the number of nodes, $\delta = \log d$, where d is the network diameter. We assume the number of objects is O(n). Both stretch and hops refer to an object search.In most cases, the time for insertion is given with high probability. Also, in some cases, various messages can be sent in parallel; we did not allow for this optimization in stating the bounds in this table.

participating nodes as well as significant work to preprocess this set to generate a routing infrastructure. Further, should nodes fail, the PRR scheme was unable to adapt to changes.

In this paper,

- We present Tapestry; a simplification of the PRR scheme for object location. While we cannot prove that Tapestry object location meets the same bounds on stretch as the PRR paper, but we note that the simplification does not appear to hurt its performance too much.

- We extend Tapestry (as well as the PRR approach) to deal with a changing participant set. We allow nodes to arrive and depart while maintaining the ability for existing objects to be located and new objects to be published. This works for a superset of the specialized metric space that PRR assumed.

- We also observe that a static version of the PRR scheme can be used for general metric spaces (i.e. spaces that do not meet the condition assumed by PRR) to get results similar to the results of Awerbuch and Peleg [1].

We note that our goals are to analyze the simple schemes that are the basis of the PRR and the Tapestry algorithms, as much as to get the best possible asymptotic results.

We present a summary of some of the previous results and our results in Table 1. We note that the result for general metrics can be improved using results of Thorup and Zwick [19] to use only $O(n \log^2 n)$ space.

1.1 Related Work

Some schemes that exhibit routing locality include Plaxton, Rajaraman, and Richa (PRR) [16] and Awerbuch and Peleg [1]. Both allow the publication and deletion of objects with only a logarithmic number of messages and both guarantee a low stretch. The PRR scheme finds objects with total latency that is within a constant factor of optimal for a specific class of network topologies. Moreover, it ensures that no node has too many directory entries. Awerbuch et al. route within a polylogarithmic factor of optimal for general network topologies. The Awerbuch scheme does not explicitly deal with load balancing, though it could perhaps be modified to do so. Unfortunately, both the PRR and Awerbuch schemes assume full knowledge of the participating nodes, or, equivalently, they assume that the network is static.

There is also an abundance of theoretical work on finding compact routing tables [2, 15, 7, 20] whose techniques are closely related to those in this paper. See [10] for a survey. A recent and closely related paper is that of Thorup and Zwick, who showed a sampling based scheme similar to that of PRR could be used to find small stretch routing tables and/or answer approximate distance queries for graphical metrics.

Most recent work on peer-to-peer networks ignores stretch. Chord [18] constructs a distributed lookup service using a logarithmic-sized routing table. Nodes are arranged into a large virtual circle. Each node maintains pointers to predecessor and successor nodes, as well as a logarithmic number of “chords” which cross greater distances within the circle. Queries are forwarded along chords until they reach their destination. CAN [17] places objects into a virtual, high-dimensional space. Queries are routed along axes in this virtual space until they reach their destination. Pastry [9] is based on the PRR scheme, routing queries via successive resolution of digits in a high-dimensional name space. However, it does not provide the routing locality of the PRR scheme. All of these schemes can find objects with a polylogarithmic number of application-level network hops, while ensuring that no node contains more than its share of directory entries. Moreover, they can support the introduction and removal of new participants in the peer-to-peer network.

Recent peer-to-peer systems can locate objects in a dynamic network. Gnutella [14] utilizes a bounded broadcast mechanism to search neighbors for documents. FreeNet [5] utilizes a chaotic routing scheme in which objects are published to a set of nearest neighbors and queries follow gradients generated by object pointers; the behavior of FreeNet appears to converge somewhat toward the PRR scheme when a large number of objects are present[2]. Neither of these techniques are guaranteed to find objects.

Table 1 summarized related work. In the table, d is the diameter of the network, and b is the base of the logarithm used in PRR's scheme; it does not depend on n, the number of nodes in the network. For the network distance results, we assume that PRR's (described in Section 3) condition holds.

1.2 Techniques

The crux of our methods for insertion and deletion of nodes into the network lie in an algorithm for maintaining nearest neighbors in our restricted metric space. Our approach follows that Karger and Ruhl [12], who give a sequential algorithm for answering nearest neighbor queries in a similarly restricted metric space. [3]

Karger and Ruhl describe two data structures, one based on sampling (also the main idea underlying the PRR scheme) which is not dynamic, and one based on using a random permutation to help maintain the sampling. The latter approach is dynamic and reminiscent of the Chord network infrastructure.

Our result can be used to give a sequential dynamic algorithm for the nearest neighbor problem that is based on the sampling approach. When viewed as such, our algorithm gives slightly better space bounds than that of Karger and Ruhl's at a corresponding cost in running time.

We also point out that an alternate scheme by Plaxton, Rajaraman, and Richa gives a low stretch solution for general metric spaces. This follows from arguments similar to those used by Bourgain [4] for metric embeddings. In particular, we show that this scheme leads to a covering of the graph by trees such that for any two nodes u and v at distance δ they are in a tree of diameter $\delta \log n$. Indeed, by modifying the PRR scheme along the lines proposed by Thorup and Zwick [19] one can improve the space bounds by a logarithmic factor. But we do not address this issue here.

1.3 Outline

The remainder of this paper is divided as follows: Then Section 2 describes the details of the Tapestry infrastructure, highlighting differences with the PRR scheme and introducing concepts and terminology for the remainder of the paper. Section 3 describes how we can solve the incremental nearest neighbor problem, Section 4 then explains how this is used as part of inserting a node. Section 5 discusses deletion. Finally,

[2]This is a qualitative statement at this time

[3]Clarkson also presented a very similar approach in [6].

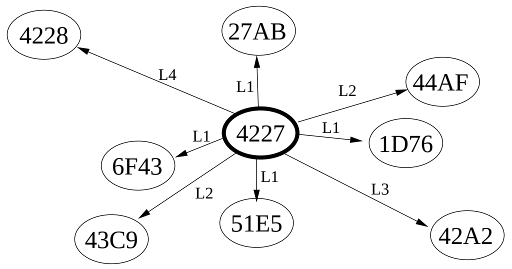


Figure 1: *Tapestry Routing Mesh.* Each node is linked to other nodes via *neighbor links*, shown as solid arrows with labels. Labels denote which digit is resolved during link traversal. Here, node `4227` has an L1 link to `27AB`, resolving the first digit, an L2 link to `44AF`, resolving the second digit, etc.

Section 6 gives a simple proof that PRR straw man scheme has polylogarithmic stretch. Section 7 then concludes.

2 The Tapestry Infrastructure

Tapestry [21] is the wide-area location and routing infrastructure of OceanStore [13]. Tapestry assumes that nodes and documents in the system can be identified with unique identifiers (names), represented as strings of digits. Digits are drawn from an alphabet of radix b. Identifiers are uniformly distributed in the namespace. We will refer to node identifiers as *node-IDs* and document identifiers as *globally unique identifiers* (GUIDs). Among other things, this means that every query has a unique destination GUID which ultimately resolves to a node-ID. For a string of digits α, let $|\alpha|$ represent the number of digits in that string.

Tapestry inherits its basic structure from the data location scheme of Plaxton, Rajaraman, and Richa (PRR) [16]. As with the PRR scheme, each Tapestry node contains pointers to other nodes (*neighbor links*), as well as mappings between object GUIDs and the node-IDs of storage servers (*object pointers*). Queries are routed from node to node along neighbor links until an appropriate object pointer is discovered, at which point the query is forwarded along neighbor links to the destination node.

2.1 The Tapestry Routing Mesh

The Tapestry *routing mesh* is an overlay network between participating nodes. Each Tapestry node contains links to a set of neighbors that share prefixes with its node-ID. Thus, neighbors of node-ID α are restricted to nodes that share prefixes with α, *i.e.* nodes whose node-IDs $\beta \circ \delta$ satisfy $\beta \circ \delta' \equiv \alpha$ for some δ, δ'. Neighbor links are labeled by their *level number*, which is one greater than the number of digits in the shared prefix, i.e. $(|\beta| + 1)$. Figure 1 shows a portion of the routing mesh. For each *forward neighbor pointer* from a node A to a node B, there will a *backward neighbor pointer* (or “backpointer”) from B to A.

Neighbors for node α are grouped into *neighbor sets*. For each prefix β of α and each symbol $j \in [0, b-1]$, the neighbor set $\mathcal{N}^{\alpha}_{\beta,j}$ contains Tapestry nodes whose node-IDs share the prefix $\beta \circ j$. We will refer to these as (β, j) neighbors of α or simply (β, j) nodes. When context is obvious, we will drop the superscript α. Let $l = |\beta| + 1$. Then, the collection of b sets, $\mathcal{N}^{\alpha}_{\beta,j}$, form the level-$l$ routing table. There is a routing table at each level, up to the maximum length of node-IDs. Membership in neighbor sets is limited by constant parameter $K \geq 1$: $|\mathcal{N}^{\alpha}_{\beta,j}| \leq K$. Further, to the extent possible, $|\mathcal{N}^{\alpha}_{\beta,j}| < K \Rightarrow \mathcal{N}^{\alpha}_{\beta,j}$ contains all (β, j) nodes.

Property 1 (Consistency) *If $\mathcal{N}^{\alpha}_{\beta,j}=\phi$, for any α, then there are no (β, j) nodes in the system. We refer to this as a "hole" in α's routing table at level $|\beta|+1$, digit j.*

Property 1 implies that the routing mesh is fully connected. Messages can route from any node to any other node by resolving the destination node-ID one digit at a time. Let the source node be α_0 and destination node be $\beta \equiv j_1 \circ j_2 \ldots j_n$. Then routing proceeds by choosing a succession of nodes: $\alpha_1 \in \mathcal{N}^{\alpha_0}_{\phi,j_1}$ (first hop), $\alpha_2 \in \mathcal{N}^{\alpha_1}_{j_1,j_2}$ (second hop), $\alpha_3 \in \mathcal{N}^{\alpha_2}_{j_1 \circ j_2,j_3}$ (third hop), *etc.*

Property 2 (Locality) *The crucial property shared by both Tapestry and the PRR scheme is that each $\mathcal{N}^{\alpha}_{\beta,j}$ contains the closest (β, j) neighbors as determined by a given metric space. The closest neighbor with prefix $\beta \circ j$ is referred to as a* primary neighbor*, while the remaining ones are* secondary neighbors*.*

Property 2 yields the important locality behavior of both the Tapestry and PRR schemes. Further, it yields a simple solution to the *static nearest-neighbor problem*: Each node α can find its nearest neighbor by choosing from the set $\bigcup_{j\in[0,b-1]} \mathcal{N}^{\alpha}_{\phi,j}$. Section 3 will discuss how to maintain Property 2 in a dynamic network.

2.2 Routing to Objects with Low Stretch

Tapestry maps each document GUID, ψ, to a set of *root nodes*: $\mathcal{R}_\psi$=MAPROOTS(ψ). We call $\mathcal{R}_\psi$ the *root set* for ψ, and each $\alpha \in \mathcal{R}_\psi$ is a *root node* for ψ. It is assumed that MAPROOTS(ψ) can be evaluated anywhere in the network.

To function properly, MAPROOTS(ψ) must return nodes that exist. The size of a root set, $|\mathcal{R}_\psi| \geq 1$, is small and constant for all documents. The simplest version of the Tapestry infrastructure utilizes $|\mathcal{R}_\psi| = 1$. In this case, we can speak of *the root node* for a given node, ψ.

Property 3 (Unique Root Set) *The root set, $\mathcal{R}_\psi$, for document ψ must be unique. In particular,* MAPROOTS*(ψ) must generate the same $\mathcal{R}_\psi$, regardless of where it is evaluated in the network.*

Storage servers *publish* the fact that they are storing a replica by routing a publish message toward each $\alpha \in \mathcal{R}_\psi$. Publish messages are routed along primary neighbor links. At each hop, publish messages deposit *object pointers* to the object. Unlike the PRR Scheme, Tapestry maintains *all* object pointers for objects with duplicate names (i.e. copies). Figure 2 illustrates publication of two replicas with the same GUID.

Queries for document ψ route toward *one* of the root nodes $\alpha \in \mathcal{R}_\psi$ along primary neighbor links until they encounter an object pointer for ψ, then route to the located replica. If multiple pointers are encountered, the query proceeds to the closest replica. At the beginning of the query, we select a root node randomly from $\mathcal{R}_\psi$. Figure 3, shows three different location paths. In the worst case, a location operation involves routing all the way to root. However, if the desired object is close to the client, then the query path will intersect the publishing path before reaching the root with high probability.

In the PRR scheme, queries route by examining all secondary neighbors before proceeding along the primary link toward the root. The number of secondary neighbors is set according to their metric space, but bounded by a constant.

Theorem 1 *PRR and Tapestry can perform location-independent routing, given Property 3.*

Proof: The publishing process ensures that all members of $\mathcal{R}_\psi$ contain mappings (object pointers) between ψ and every server which contains ψ. Thus, a query routed toward any $\alpha \in \mathcal{R}_\psi$ will (in the worst case) encounter a pointer for ψ after reaching α. ■

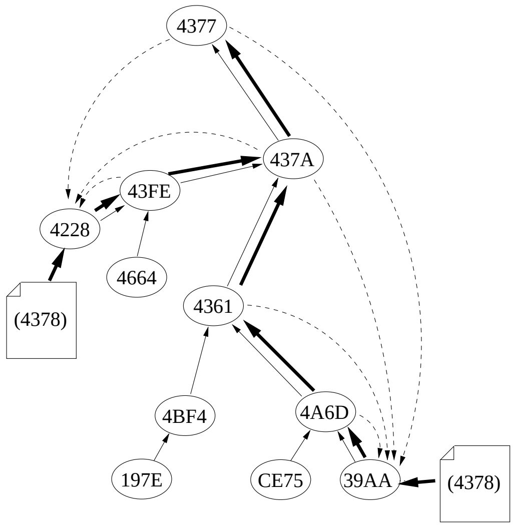


Figure 2: *Publication in Tapestry.* To publish document `8734`, server `39AA` sends publication request towards the root, leaving a pointer to itself at each hop. Server `8224` publishes its replica similarly.

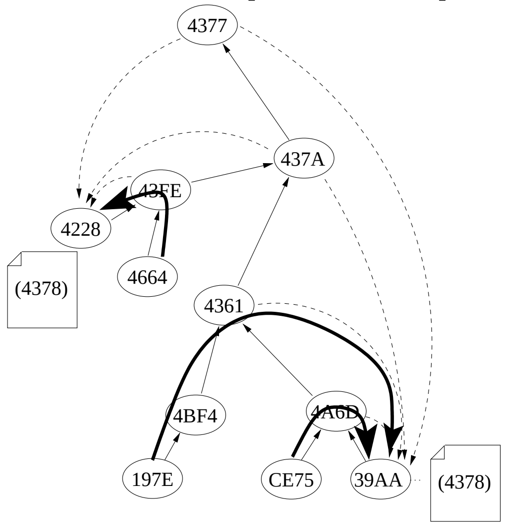


Figure 3: *Routing in Tapestry:* Three different location requests. For instance, to locate GUID `8734`, query source `197E` routes towards the root, checking for a pointer at each step. At node `1634`, it encounters a pointer to server `1634`.

Observation 1 *(Fault Tolerance) If $|\mathcal{R}_\psi| > 1$ and the names in $\mathcal{R}_\psi$ are unrelated to one another, then we can retry document queries and tolerate a number of faults in the Tapestry routing mesh.*

In a general metric space, it is difficult to make claims about the performance of such a system. PRR restrict their attention to metric spaces that have a certain even-growth property. In particular, they assume that for a given point A, the ratio of the number of points within $2r$ of A and the number of points within distance r of A is bounded above and below by constants. (Unless all points are within $2r$ of A.) Given this constraint, [16] shows the average distance traveled in locating an object is *proportional* to the distance from that object, *i.e.* queries exhibit $O(1)$ stretch. Tapestry is a simpler system and seems to provide low stretch in practice.[21]

2.3 Surrogate Routing

The procedures for *publishing* and *querying* documents outlined in Section 2.2 do not require the actual membership of $\mathcal{R}_\psi$ to be known. All that is required is to be able to compute the next hop toward the root from a given position in the network. As long as this *incremental* version of MAPROOTS() is consistent in its behavior, we achieve the same routing and locality behavior as in Section 2.2. Assume that INCRMAPROOTS(ψ,β) produces an ordered list of the next hop toward the roots of ψ from node β.

In the PRR scheme, MAPROOTS(ψ) produces a single root node α which matches in the largest possible number of prefix bits with ψ. Ties are broken by consulting a global order of nodes. The PRR scheme specifies a corresponding INCRMAPROOTS() function as follows: the neighbor sets, $\mathcal{N}$, are supplemented with additional *root links* that fill holes in the routing table. To route a message toward the root node, PRR routes directly to ψ as if it were a node in the Tapestry mesh. When this process encounters a hole in the neighbor table[4], it continues for one additional hop along one of these preconfigured "root links". Assuming that the supplemental root links are consistent with one another, every publish or query for document ψ will head toward the same root node.

We call the above process *surrogate routing*, since it involves routing toward ψ as if it were a node, then adapting when the process fails. Roots generated in this way are considered *surrogate roots* of ψ.

In a dynamic network, maintenance of these additional pointers can be tricky, since they follow from a "global order". Tapestry utilizes a slightly different scheme that relies on information local to each node and already present in the routing table. Rather than filling holes in the neighbor tables, we route around them. When there is no match for the next digit, we route to the next filled entry in the same level of the table, wrapping around if needed. For example, if the next digit to be fixed was 3, and there was no entry, try 4, then 5, and so on. When the routing can go no further (the only node left in the current table level is the current node), that node is the root. This scheme is simpler than the PRR scheme under inserts and deletes, and may have better load balancing properties.

Theorem 2 *Suppose Property 1 holds. Then the Tapestry version of surrogate routing will produce a unique root.*

Tapestry's surrogate routing mechanism may introduce additional hops over PRR; however, the number of additional hops is independent of n and expected to be less than 2 [21].

Observation 2 *(Multiple Roots) Surrogate routing generalizes to multiple roots. First, a pseudo-random function is employed to map the initial document GUID ψ into a set of identifiers ψ_0, ψ_1,. . .ψ_n. Then, to route to root i, we surrogate route to ψ_i.*

[4]This is highly likely, since the node name space is sparse.

```
   method AcquireNeighborTable (NewNodeName,NewNodeIP, PSurrogateName,PSurrogateIP)
1  $\alpha \leftarrow$ GreatestCommonPrefix(NewNodeName, PSurrogateName)
2  maxLevel $\leftarrow$ Length($alpha$)
3  startList $\leftarrow$ AcknowledgedMulticast (PSurrogateIP, $\alpha$, SendID[NewNodeIP, NewNodeName])
4  BuildTableFromList(startlist, maxLevel)
5  for  i = maxlevel - 1 to  0
6        list $\leftarrow$ GetAllLevels(neighbors[$i + 1$], $i$)
7        BuildTableFromList(list, i)
   end AcquireNeighborTable

   method GetNextList (neighborlist, level)
1  nextList $\leftarrow \emptyset$
2  for $n \in$ neighborlist
3       temp $\leftarrow$ GetForwardAndBackPointers($n$, level))
4       nextList $\leftarrow$ KeepClosestk(temp $\cup$ nextList)
5  return  nextList    end GetNextList
```

Figure 4: Building a Neighbor Table

3 Building Neighbor Tables

Though building the neighbor table is not the first step in the insertion process, it is the most complex and interesting step, so we discuss it first. We want to build the neighbor sets, $\mathcal{N}^{\alpha}_{\beta,j}$ for a new node α. These sets must adhere to Properties 1 and 2. This amounts to solving the nearest neighbor problem for many different prefixes. A recent paper by Karger and Ruhl [12] provides a method for solving the nearest neighbor problem in the same metric space as we consider here. The method we present below has worse time bounds than their results, but requires no additional space over the PRR data structures.

As in [16], we make the following constraint on the network. Let $\mathcal{B}_A(r)$ denote the all points within r of A, and $|\mathcal{B}_A(r)|$ denote the number of such points. We assume that

$$|\mathcal{B}_A(2r)| \leq c\,|\mathcal{B}_A(r)|\,, \tag{1}$$

for some constant c. PRR also assume that $|\mathcal{B}_A(2r)| \geq c\,|\mathcal{B}_A(r)|$, but we will not need that. Notice that our expansion property is exactly that used by Karger and Ruhl [12].

Figure 4 shows how this is done. We start with a list of all nodes matching some prefix; we get this list via a multicast. Then generate a new list consisting of all the nodes known by the nodes on the first list matching the next prefix; that is, both forward neighbor links and backpointers. We then trim this list to include only the closest k nodes, and continue, until we find the closest k nodes with the empty prefix.

We then use these lists to fill in the neighbor table. In particular, to fill in level i of the neighbor table, we look in the level-i list. For $j \in [0, b-1]$, we keep the closest node with prefix $\alpha_i \circ j$. If $k = O(\log n)$, then with high probability, we know there is one (or indeed, any constant) number of such nodes on the list for every j.

Theorem 3 *If the c is the expansion constant of the network and $c^2 \leq b$ (where b is the digit size), then the algorithm of figure 4 will produce the correct neighbor table with high probability.*

The new node also induces changes on other nodes neighbor tables. In Theorem 4, we prove that any node needing to update its level i link is one of the closest k nodes for level-i with high probability. We

```
  method INSERT (gatewayIP, NewNodeIP, NewNodeName )
1 (PSurrogateIP, PSurrogateName) $\leftarrow$ ACQUIREPRIMARYSURROGATE (gatewayIP, NewNodeName)
2 $\alpha \leftarrow$ GREATESTCOMMONPREFIX(NewNodeName, PSurrogateName)
3 ACKNOWLEDGEDMULTICAST (PSurrogateIP, $\alpha$, LINKANDTRANSROOT[NewNodeIP, NewNodeName])
4 ACQUIRENEIGHBORTABLE (NewNodeName,NewNodeIP,PSurrogateIP, PSurrogateIP)
  end INSERT
```

Figure 5: Node Insertion Routine. The insertion process begins by contacting a gateway node, which is a member of the Tapestry network. It then transfers object pointers and optimizes the neighbor table.

```
  method ACKNOWLEDGEDMULTICAST($\alpha$,FUNCTION)
1 apply FUNCTION

2 if NOTONLYNODEWITHPREFIX($\alpha$)
3    for $i = 0$ to $b - 1$
4        neighbor $\leftarrow$ GETMATCHINGNEIGHBOR($\alpha \circ i$)
5        if neighbor exists
6           $\mathcal{S} \leftarrow$ ACKNOWLEDGEDMULTICAST (GETIP(neighbor) $\alpha \circ i$, FUNCTION )
7 wait $\mathcal{S}$
8 SENDACKNOWLEDGEMENT()
  end ACKNOWLEDGEDMULTICAST
```

Figure 6: Acknowledged Multicast

assume here that we only need one neighbor, but it should be clear how to extend this proof to handle any constant number.

Theorem 4 *If node A has a primary forward pointer to B (so A is the closest node to B with prefix $\alpha \circ j$), then with high probability, A is among the $k = O(\log n)$ closest nodes to B.*

Since each node has an expected constant number of pointers, the expected time of this algorithm is $O(k) = O(\log n)$ per level or $O(\log^2 n)$ overall.

The number of backpointers is less than $O(\log n)$ with high probability, so we get a total time of $O(\log^3 n)$ with high probability. But one can do better. Using the techniques of Theorems 3 and Theorem 4, one can argue that with high probability, all the visited level i nodes are within a ball of radius $4\delta_{i+1}$. Further, with high probability, there are only $O(\log n)$ level i nodes within $4\delta_{i+1}$. This means we visit only $O(\log n)$ nodes per level, or $O(\log^2 n)$ nodes overall.

Furthermore, notice that $\delta_i \leq \frac{1}{3}\delta_{i+1}$. Suppose the number of nodes touched at each level is bounded by q. We know (by the above argument) that $q = O(\log n)$. The total network traffic is bounded by

$$\sum_i \delta_i q = q \sum_i \delta_i$$

But since the δ_i are geometrically decreasing, they sum to something to $O(d)$, so the total network traffic is $O(qd) = O(d \log n)$.

4 Node Insertion

In this section, we will describe the overall insertion algorithm, using the nearest neighbor algorithm as a subroutine. We would like the network after the insertion to be the same as if we had been able to build the network from static data. This means maintaining the following invariant.

Property 4 *If node A is on the path between a publisher of object o and the root of object o, then A has a pointer to object o.*

In this section, we will show that if Property 1, Property 2, and property 4 hold, then we can insert a node such that all three after the insertion, and the new node is part of the network. It may, however, happen that during a node insertion, one or both of the properties is temporarily untrue. In the case of Property 1, this can be particularly serious since some objects may become temporarily unavailable. Section 4.3 will show how the algorithm can be extended to eliminate this problem.

Figure 5 shows the basic insertion algorithm. First, the new node contacts the closest matching node. Then the node contacts the subset of nodes that must be notified to maintain Property 1. These are the nodes that have a hole in their neighbor table that the new node should fill. We use the function ACKNOWLEDGEDMULTICAST (detailed Section 4.1) to do this. As a final step, we build the neighbor tables, as described before. Notice that we can use the multicast in step 3 of the insertion algorithm to get the **startList** of the nearest neighbor table algorithm.

We would also like to maintain Property 4. This means that all nodes on the path from an object's server to the object's root have a pointer to that object. Once again, there are two cases, one where not fixing the problem means that the network may return the incorrect answer, and one where not fixing the problem makes the network slow.

First, the function LINKANDTRANSROOT from Figure 5 transfers any object pointers that should be rooted at the new node, and deletes any pointers that should not now be on the current node. If we do not move the object pointers, then objects may become unreachable. For performance reasons, while building the neighbor table, any node that adds the new node as a primary neighbor requests a republish for any object that would have gone through the new node.

4.1 Acknowledged Multicast

To contact all nodes with a given prefix we introduce an algorithm we call Acknowledged Multicast. The algorithm is shown in Figure 6.

To be valid, the prefix α must be a prefix of the receiving node. When a node receives a multicast message for prefix α, it sends the message to one node with each possible extension of α; that is, for each j, it sends the message to one node with prefix $\alpha \circ j$ if such a node exists. We know by Property 1 that if an $\alpha \circ j$-node exist, then every α-node knows at least one such node. Each of these nodes then continues the multicast.

Because we need to know when the algorithm is finished, we also require each recipient to send an acknowledgment to its parent after receiving acknowledgments from all its children. If a node has no children, then it sends the acknowledgment immediately. When the node starting the multicast gets an acknowledgment from each of its children, we know that all nodes with the given prefix have been contacted.

Theorem 5 *When a multicast recipient sends the acknowledgment, all the nodes with the given prefix have been reached.*

These messages form a tree. If there are k nodes reached in the multicast, there are $k - 1$ edges in the tree. Alternatively, each node will only receive one multicast message, so there can be no more than $O(k)$

```
  method FixObjectPtrs (sender,deletedNode,objPtr)
1     ContinuePublish(sender,deletedNode,objPtr)
2     oldsender ← GetOldSender(objPtr)
3     if oldsender = sender
4        TellSenderToDeletePtr(sender,deletedNode,objPtr)
  end FixObjectPtrs
```

Figure 7: FixObjectPtrs

such messages sent. Each of those links could be the diameter of the network, so the total cost of a multicast to k nodes is $O(dk)$.

It is possible to build a simple variant of this algorithm where acknowledgments and a little additional information are sent to the originating node. With this modification, the multicast algorithm requires less time and no state on the intermediate nodes.

4.2 Fixing Object Pointers

This is a special partial version of republish that maintains Property 4. This function is used to rearrange the object pointers any time a node changes its primary neighbor. This is not necessary for correctness, but does ensure the network performs better.

In essence, the node making a change sends the object pointer up the new path. From the place where the old path and new path meet, a delete message is sent back down the tree. This requires maintaining a last-hop pointer for each object pointer. When the republish message reaches a node by a different path than the original publish did, a delete message is sent down the old path.

If the node uses an ordinary republish, (simply sending the message towards the root), it could leave some object pointers dangling until the next time out. For example, if eliminating node A makes the path from some object to its root skip node B, then node B will still be left with a pointer to the object.

Notice, however, that Property 4 is not critical to the functioning of the system. If a node should use FixObjectPtrs but does not, then performance suffers, but because the root node still has the pointers, the objects will still be available. Further, timeouts and regular republishes will eventually ensure that the object pointers are in the right place.

4.3 Keeping Objects Available

In this section, we explain how the above algorithm can be extended to keep objects available even during the insertion process. To do this, we prevent an object from having two roots at the same time.

During the time a node is inserting itself, an object request that would go to the new node after insertion may either go to the new node or to its pre-insertion destination. To keep objects available, if either of those two nodes receives a request for an object it does not have, it must be able to forward the request to the other node.

If an inserting node receives an object request for an object it does not have, it sends the object request back out, routing as if it did not know about itself. That is, if the new node fills a hole at level i, it sends out a message with the level at $i + 1$ to one of the surrogate nodes. (This is possible even without a neighbor table; the new node need only know the identity of one surrogate.) The surrogate then routes the message as it would have if the new node had not yet entered the network.

If a surrogate node receives an object request for an object pointer that is has already send to the new node, it needs to forward it on to the new node. But we want to do this in such a way that the surrogate

```
   method ObjectNotFound (objectID)
1  if (Inserting)
2     level ← length(GreatestCommonPrefix(NewNodeName, PSurrogateName))
3     Route(objectID,PSurrogateName, level)
4  elseif not RoutingConsistentWithNeighbors(objectID)
5     RetryRouting(objectID,Neighbors)
6  endif
   end ObjectNotFound
```

Figure 8: Misrouting and route correction to maintain object availability

does not need to keep any state to show which nodes are inserting. So we require all nodes to "check the routing" of an object request or publish before rejecting it. By this, we mean that the nodes test whether the object made a surrogate step that it did not need to make. If it finds out it did make a surrogate step instead of going to the new node, the current node redirects the message to the new node.

To make this work properly, we require that the old root not delete pointers until the new root has acknowledged receiving them. If this is done, then this system always finds the object. If an object arrives at the old root before the old root transfers pointers, everything works as if the new node did not exist. If an object arrives at the old root after the the node has move pointers, then the old root must clearly know about the new node, and so is able to forward the object request on, and the new node services it.

If a request for an object arrives at the new node, and the new node does not yet have the pointer, the new node forwards it on to the old root. If the old root does not know about the new node, it must have a copy of the object pointer. If the old root does know about the new node, then it would have forwarded on the object pointers already. If the request arrives at the new node after it gets the object pointers, then all is well.

It is possible for a request for a non-existent object to loop until the insertion is complete; we can get around this by including information in the message header about where the request has been.=

4.4 Network Traffic

Ignoring objects, the total network traffic required for node insertion is $O(d \log b)$ with high probability, where d is the diameter of the network. The total number of hops is $O(\log^2 n)$ with high probability. The first step is no more costly than searching for an object pointer, and [16] argues that finding an object pointer requires $O(d)$ messages. The multicast takes time $O(kd)$ where k is the number of nodes reached. But k will be small and a constant relative to n. Finally, building the neighbor tables takes $O(\log^2)$ messages.

If there are m objects that should be on the new node, then the cost of republishing all those object is at most $O(md)$. This give a total traffic of $O(md \log n)$ for object pointer moves.

4.5 Simultaneous Insertion

In a wide-area network, insertions will not happen one at a time. If two nodes are inserted at once, each may get an older view of the network, so neither node will see the other. Suppose A and B are inserted simultaneously. There are three possibilities:

- A's and B's insertions do not intersect. This is the most likely case; A need only know about $O(\log^2 n)$ nodes with high probability so the chance that B is one of them is small.

- For some (α, j), B should be one of the (α, j) neighbors of A, but A has some further (α, j) neighbor instead.

- For some (α, j), B is the only possible neighbor.

In the first case nothing needs to be done. In the second case, if B fails to get added to A's neighbor table, then the network still satisfies all object requests, but the stretch may increase. By reinserting a node after a random amount of time, we can ensure that this small problem is only temporary.

The third case is a much greater cause for concern, since if A has a hole where B should be, Property 1 would no longer hold. This could mean that some objects become unavailable. This problem can also be solved by reinserting the node, but before the reinserting occurs, objects may be unavailable, and this is a serious problem we wish to avoid.

We will now argue that if we can serialize the multicasts, the rest of the insertion algorithm is easy. That is, suppose A and B both insert themselves at once. Let their common prefix be α. If A's multicast reaches the α-nodes common to both their multicasts before B's does, then A will know of B. (The same is obviously true if we reverse A and B.) To see this, notice that A is no different than a node completely in the network for the purposes of B's multicast. It is also the case that B will able to find A during its neighbor table building phase, since all the necessary nodes have forward pointers to A. This is, in fact, the only reason we use the forward pointers in the neighbor table building algorithm.

But what happens some α-nodes receive A's multicast before B's and other receive B's multicast before A's? In this case, it is possible that neither node will notify the other. (Notice that A and B may be multicasting to different prefixes; A's multicast prefix could be α, while B's could be $\alpha \circ j$ for some j, but this does not affect the argument.)

To solve this problem, we slightly modify the multicast algorithm so that it reaches inserting nodes.

One solution to this problem would be to keep a list of recent multicast messages. We reject this solution because it raises a number of problems. First, it requires keeping state that is not used in the common case. Second, it is not clear how long the time out should be. If it is too short, then keeping the extra state is pointless, but it is not workable to keep the state forever.

Instead, each node sends down a "wish list" of prefixes it could not reach. (Note that this list for a given level can be compactly represented as a bit vector of length b, where each bit indicates whether a node with that digit was contacted or not.) Any receiving nodes checks the wish list to see if it can reach any node on the list. If it can, then it sends the multicast to the new node, adjusting the prefix and hole pattern to what they would have been had the new node been contacted at the right time.

The new version is shown in the appendix, in Figure 13. In addition to the information passed before, this new multicast includes the set of expected neighbor table holes. If any node receiving the multicast notices that a hole has been filled, it sends the multicast message (adjusting the prefix and hole table as needed) to a node in the former hole.

This can result in many multicast messages to a new node. However, this is not a burden. First, this is the uncommon case. Second, it is rare that the new node will be anything other than a leaf in the tree (i.e. the new node will not have to forward the multicast). Finally, the new node can suppress duplicate multicast messages. While this would be burdensome for an established node, keeping a little extra state during insertion is not unreasonable.

Doing this guarantees the following fact:

Theorem 6 *Suppose A and B both insert. If there is a node that receives A's multicasts before B's, and some other node that receives B's multicast before A's, then A will receive B's multicast if A should receive B's multicast.*

The proof is in the appendix Section 6.

```
method DeleteSelf ()
1       SendPointersToSurrogates()
2       for $level = 0$ to $maxLevel$
3           SendReplacement(backpointers,level, GetNearest(neighbors, level))
4       LeavingNetwork(selfID,neighbors)
5       for $o \in objectPointers$
6           FixObjectPointers(o)
end DeleteSelf
```

Figure 9: Node Disintegration

5 Delete

In this section we will talk about how nodes leave the network. We consider two cases: *voluntary* and *involuntary* delete. A voluntary delete occurs when a node informs the network that it is about to exit. This is clearly an optimistic situation that permits fixup of neighbor links and object pointers. An involuntary delete occurs when a node ceases to function without warning.

5.1 Voluntary Delete

A deleting node should remove itself in such a way that object location can continue seamlessly. When the node A leaves the network, several things happen. A node or set of nodes will become roots for objects currently rooted at A. Also, some objects whose paths to the root went through A will have use different paths.

The first step is to move the object pointers for which the exiting node is the root. This needs to be done via a multicast, since the exiting node does not have direct knowledge of its surrogates. Note, however, that if we are willing to tolerate objects being temporarily unavailable, this step can be skipped.

The second step is notify neighbors that the node is leaving the network. By itself, this could leave neighboring nodes with a hole in their routing table. The nodes would not know if the table was empty because no node exists to fill it or not, potentially breaking Property 1. To prevent this, as part of the "I'm leaving" message described above, for each level l, the exiting node sends the closet level l node. Given that information, the other nodes can maintain their neighbor tables. (The neighbor tables will not be perfect after such a deletion, but nodes that need perfect neighbor tables may always reinsert themselves.)

As a final step, the exiting node republishes all objects on that node using FixObjectPointers as described in Section 4.2.

5.2 Involuntary Delete

It will not always be possible for a node to delete itself, so a practical network should handle unexpected deletes. We propose that unexpected deletes be handled lazily. That is, when a node notices some other node is down, it does everything it can to fix its own state, but does not attempt to fix the state of any other node.

A deletion will not be noticed until some node tries to send a message to the now-defunct node and does not get a response. When that happens, the sending node should first remove the node from its neighbor table. If this produces a hole in the table, it will have to do additional work to ensure Property 1 is maintained. Second, it should republishOnDelete all object pointers that would have gone through the new node.

```
  method DELETEOTHER (deadNode)
1     createHole ← REMOVEFROMNEIGHBORS(neighbors, deadNode)
2     if createHole
3         $\alpha$ ← GREATESTCOMMONPREFIX(SelfName, DeadNode)
4         $\beta$ ←DeadNode[LENGTH($\alpha$)+1]

5         ACKNOWLEDGEDMULTICAST ($\alpha$, REQUESTNODE[$\beta$])
6     for objptr in objectPointers
7     if USEDDELETEDNODE(objptr,DeadNode)
8         REPUBLISHONDELETE(deletedNode,objPtr)
  end DELETEOTHER
```

Figure 10: DeleteOther

To ensure Property 1, if deleting the node leaves a hole in the routing table, A performs a multicast to all nodes sharing the same prefix as A and the dead node. If none of those nodes knows of a node to fill the hole, then A assumes the hole cannot be filled, and should inform all the nodes touched in the multicast. Likewise, if A does find a node to fill the hole, it also informs the other nodes. Partial psuedocode is found in Figure 10.

There are two problems with this scheme. The first problem is that the reinsertion may take a long time and it is not clear what to do with object requests in the mean time.

The second practical problem could be more serious. If a node becomes unreachable from one part of the network but is still reachable from the other, then part of the network will delete the node, and part of it will not. This problem would never be detected.

It may be that two nodes notice the disappearance at about the same time and that both nodes send out multicast messages. However, the only problem with multiple multicast messages is unnecessary message traffic.

6 Object Location in general metric spaces.

For any metric space S, we show a way to route to an object such that the stretch is polylog with $O(|\mathrm{ID}|\log^2 n)$ average space, where $|\mathrm{ID}|$ is the size of an object ID, or $O(\log n)$. We remark that this is the strawman scheme proposed by Plaxton, Rajaraman, and Richa [16] without load balancing, and is quite similar to the scheme of Thorup and Zwick [19].

Let $S_{i,j}$ be a set of 2^i randomly chosen points in the metric space, and let $i \in [0, \log n]$ and $j \in [0, c\log n]$. Each node in the network stores the closest node in $S_{i,j}$ for each pair i, j. Also, each node in $S_{i,j}$ stores a list of all network nodes which point to it.

Suppose a node X wants to find an object Y. Then starting with $i = \log n$, X asks (for all j in parallel) its representative in the set $S_{i,j}$ if it points to Y. If one of them does, it returns the pointer to Y. If this fails, it tries $S_{i-1,j}$ for all j. Notice that there is one node in $S_{0,j}$.

Theorem 7 *Let i^* be the largest i such that there is some $S_{i,j}$ that points to both X and Y. We will show that $d(S_{i*,j}, X) \leq \log n * d(X,Y)$ with high probability. Moreover, the average space used by the data structure is $O(\log^2 n)$.*

The proof in the appendix section A.1.

To load balance, we let i range over all possible ID prefixes, and only search is that are prefixes of Y's ID. This results in a very large table size. We do not know how to efficiently maintain this data structure.

7 Conclusions

One of the most important aspects of a peer-to-peer routing system is the ability to adapt to a changing set of participants. We illustrate how to adapt to arriving and departing nodes in Tapestry, a location-independent routing infrastructure. This adaptation involves an efficient, distributed solution to the nearest-neighbor problem as well as a distributed algorithm for maintaining a prefix-based routing mesh. Both of these are presented for the first time in this paper. The cost of our integration algorithms is similar to that provided by other systems that do not provide routing locality. The result is an infrastructure that provides deterministic location, routing locality, and load balance, even in a changing network. This is a unique set of properties.

References

[1] AWERBUCH, B., AND PELEG, D. Concurrent online tracking of mobile users. In *SIGCOMM* (1991), pp. 221–233.

[2] AWERBUCH, B., AND PELEG, D. Routing with polynomial communication-space trade-off. *SIAM Journal on Discrete Mathematics 5*, 2 (1992), 151–162.

[3] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *Proc. of SIGMETRICS* (2000), ACM, pp. 34–43.

[4] BOURGAIN, J. On lipschitz embedding of finite metric spaces in hilbert space, 1985.

[5] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability* (New York, 2001), H. Federrath, Ed., Springer.

[6] CLARKSON, K. L. Nearest neighbor queries in metric spaces. In *Proc. 29th Symp. Theory Comput.* (1997).

[7] COWEN. Compact routing with minimum stretch. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)* (1999).

[8] DRUSCHEL, P., AND ROWSTRON, A. Past: Persistent and anonymous storage in a peer-to-peer networking environment. Submission to ACM HOTOS VIII, 2001.

[9] DRUSCHEL, P., AND ROWSTRON, A. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Submission to ACM SIGCOMM, 2001.

[10] GAVOILLE, C. Routing in distributed networks: Overview and open problems. *ACM SIGACT News - Distributed Computing Column 32*, 1 (Mar. 2001), 36–52.

[11] GOPAL, B., AND MANBER, U. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of ACM OSDI* (New Orleans, Louisiana, February 1999), ACM, pp. 265–278.

[12] KARGER, D., AND RUHL, M. Find nearest neighbors in growth-restricted metrics. In *ACM Symposium on Theory of Computing* (Montral,Qubec,Canada, 2002).

[13] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS* (November 2000), ACM.

[14] ORAM, A., Ed. *Peer-to-Peer, Harnessing the Power of Disruptive Technologies*. O'Reilly Books, 2001.

[15] PELEG, D., AND UPFAL, E. A tradeoff between size and efficiency for routing tables, 1989.

[16] PLAXTON, C. G., RAJARAMAN, R., AND RICHA, A. W. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM SPAA* (June 1997), ACM.

[17] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A scalable content-addressable network. In *Proceedings of SIGCOMM* (August 2001), ACM.

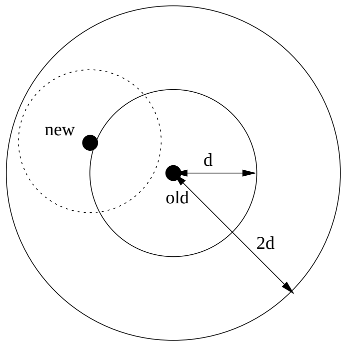


Figure 11: The larger ball around A contains $O(\log n)$ nodes, while the smaller ball contains none.

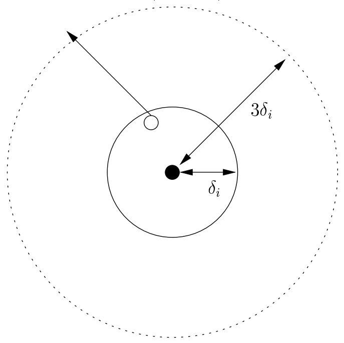


Figure 12: If δ_i is less than δ_{i+1}, then A must point to a node within r_i.

[18] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM* (August 2001), ACM.

[19] THORUP, M., AND ZWICK, U. Approximate distance oracles. In *STOC* (2001), pp. 183–192.

[20] THORUP, M., AND ZWICK, U. Compact routing schemes. In *SPAA* (2001), pp. 1–10.

[21] ZHAO, B. Y., KUBIATOWICZ, J. D., AND JOSEPH, A. D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, University of California at Berkeley, Computer Science Division, April 2001.

A Appendix

A.1 General Metric Spaces

Recall that $S_{i,j}$ is a set of 2^i randomly chosen points in the metric space, with $i \in [0, \log n]$ and $j \in [0, c\log n]$. Each node in the network stores the closest node in $S_{i,j}$ for each pair i, j. Call these pointers outpointers. Each node in $S_{i,j}$ stores a list of all network nodes which point to it. Call these pointers inpointers. The average space is then $O(|\mathrm{ID}| \log^2 n)$, though the nodes in $S_{\log n,j}$ store $O(n)$ pointers. To see this, notice that a node the number of in-pointers equals the number of out-pointers, so the total storage is $2|\text{out-pointers}|$ which is $2c\log n$.

Suppose a node X wants to find an object Y. Then starting with $i = \log n$, X asks (for all j in parallel) its representative in the set $S_{i,j}$ if it points to Y. If one of them does, it returns the pointer to Y. If this fails, it tries $S_{i-1,j}$ for all j.

Theorem A.1 (Theorem 7) *Let i^* be the largest i such that there is some $S_{i,j}$ that points to both X and Y. We will show that $d(S_{i*,j}, X) \leq \log n * d(X,Y)$ with high probability.*

Proof:

Let $\mathcal{B}_X(r)$ be the ball around X of radius r, that is, all the nodes with in distance r of X.

Now, consider a sequence of radii such that $r_k = kd$ for $k \in [1, \log n]$. If $|B(X, r_k) \cap B(Y, r_k)| \geq \frac{1}{2} |\mathcal{B}_X(r) \cup \mathcal{B}_Y(r_k)|$ we call r_k good. We now show that if there exists a good r_k the theorem holds.

Let $r = r_k$ be a good radius. Then consider i such that $2^{\log n - i} \leq |\mathcal{B}_X(r) \cup \mathcal{B}_Y(r)| \leq 2^{\log n - i+1}$. If $|\mathcal{B}_X(r) \cap \mathcal{B}_Y(r)|$ is 1/2 of $|\mathcal{B}_X(r) \cup \mathcal{B}_Y(r)|$. Notice that given a j, with constant probability, there will exactly one member of $S_{i,j}$ in the intersection and no other member in the union. We can view each j as a trial, and since we have $c \log n$ trials, with high probability, at least one will succeed. And if there is $s \in S_{i,j}$ that points to both X and Y, when X queries s, X will get a pointer to Y, so $i^* = i$.

We will now argue that you cannot have $\log n$ bad r_k. Suppose that r_k is bad. Then $|B(X, dk) \cap B(Y, dk)|$ is less than $\frac{1}{2}$of $|\mathcal{B}_X(dk) \cup \mathcal{B}_Y(dk)|$. Notice that $B(X, kd) \cap B(Y, kd)$ contains $|\mathcal{B}_X((k-1)d) \cup \mathcal{B}_Y((k-1)d)|$, and since $|\mathcal{B}_X(kd) \cup \mathcal{B}_Y(kd)| \geq 2\,|B(X, kd) \cap B(Y, kd)| \geq 2\,|\mathcal{B}_X((k-1)d) \cup \mathcal{B}_Y((k-1)d)|$. But this can happen at most $\log n$ times, since $|B(X, r_1) \cap B(Y, r_1)| \geq 2$ (since it contains X and Y) and the network has only n nodes.

Finally, if at any point $|\mathcal{B}_X(r_k) \cup \mathcal{B}_Y(r_k)|$ contains the whole network, then let $i^* = 0$, and since there is only one element of each $S_{0,j}$, so they will clearly be pointing to both X and Y. ■

Finally, notice that if $d(S_{i*,j}, X) \leq \log n * d(X,Y)$, the total distance traveled on level i is $\log^2 n d(X,Y)$, and the latency (waiting time) is $\log n d(X,Y)$. Since there may be $\log n$ levels, this means the total latency is proportional to $d(X,Y)\log^2 n$ and total distance traveled proportional to $c * d(X,Y)\log^3 n$.

A.2 Surrogate Routing

Theorem A.2 (Theorem 2) *Suppose Property 1 holds. Then the Tapestry version of surrogate routing will produce a unique root.*

Proof: Proof by contradiction. Suppose that messages for an object with ID X end routing at two different nodes, A and B. Let β be the longest common prefix of A and B, and let i be the length of β. Then, let A' and B' be the nodes that do the $i+1$st routing step; that is, the two nodes that send the message to different digits. Notice that after this step, the first $i+1$ digits of the prefix remain constant in all further routing steps. Both $\mathcal{N}^a_{\beta,*}$ and $\mathcal{N}^b_{\beta,*}$ must have the same pattern of empty and non-empty entries. That is, if $\mathcal{N}^a_{\beta,j}$ is empty, then $\mathcal{N}^b_{\beta,j}$ must also be empty, or Property 1 is untrue. So both A' and B' must send the message on a node with the the same $i+1$th digit, so this is a contradiction. ■

A.3 Building Neighbor Tables

Theorem A.3 (Theorem 3) *If the c is the expansion constant of the network and $c^2 \leq b$ (where b is the digit size), then the algorithm of figure 4 will produce the correct neighbor table with high probability.*

Proof: We must show that given the k closet level $i+1$ nodes, we can find the the k closest level i nodes. Let δ_i be the radius of the smallest ball around the new node containing k level i matches. We would like to show that any node A inside the ball must point to a level $i+1$ node within r_{i+1} of the new node. If that is the case, then we will query A's parent, and so find A itself.

If $k = \Omega(\log n)$, with high probability there is at least one level-i node that is also a level-$(i+1)$ node, so the distance between A and its nearest level-$(i+1)$ node is no more than $2\delta_i$, since both A and the level-$(i+1)$ node are within the ball of radius δ_i. By the triangle inequality, the distance between the new

```
  method AcknowledgedMulticast($\alpha$,Function, **holepattern**)
1 apply Function
2 MulticastToFilledHoles(**holepattern**)
3 if NotOnlyNodeWithPrefix($\alpha$)
4    **moreholes** $\leftarrow$ GetHoles(**neighbors**,$\alpha$)
5    for $i = 0$ to $b - 1$
6       *neighbor* $\leftarrow$ GetMatchingNeighbor($\alpha \circ i$)
7       if *neighbor* exists
8          $\mathcal{S} \leftarrow$ AcknowledgedMulticast (GetIPneighbor $\alpha \circ i$, Function **holepattern** $\circ$ **moreholes** )
9 wait $\mathcal{S}$
10 SendAcknowledgement()
  end AcknowledgedMulticast
```

Figure 13: Acknowledged multicast with a the hole pattern

node and the node A points to is no more than $2\delta_i + \delta_i = 3\delta_i$. (See Figure 12.) This means that as long as $3\delta_i < \delta_{i+1}$, A must point to a node inside δ_{i+1}.

Now, we must show that $3\delta_i < \delta_{i+1}$ with high probability. Let l be the number of nodes such that the one expects $(1-\lambda)^{-1}k$ level i nodes. Where λ is chosen small enough such that $(1-\lambda)b/c^2 > 1$. Now let l' be the size of the ball containing k nodes. We consider two cases. In the first, the inner ball is has too high a concentration of level i nodes; in the second, the outer ball has too low a concentration.

case 1 If $l' \geq l$. That means that the ball containing l' contains k nodes. Let X_m be a random variable representing the number of level-$(i+1)$ nodes in m trials. Then we wish to bound $Pr[X_{l'} \leq k]$. But $Pr[X_{l'} \leq k] \leq Pr[X_l \leq k]$, since $l \leq l'$. But $Pr[X_l \leq k] = Pr[X_l \leq (1-\lambda)E[E_l]]$.

case 2 If $l' \leq l$, then consider the ball of radius $3\delta_i$ around the new node. This ball must contain k level-$(i+1)$ nodes (δ_{i+1} is bigger that $3\delta_i$), so the ball of radius $4\delta_i$ must also contain k level-$(i+1)$ nodes. Further, we know the volume of this ball is less than c^2l'. Let Y be the number of level $i+1$ nodes, then $Pr[Y_{c^2l'} \geq k] \leq Pr[Y_{c^2l} \geq k]$, and this is the same as $Pr[Y_{c^2l} \geq (1-\lambda)b/c^2E[Y_{c^2l}]]$, by Equation 1. We know that $(1-\lambda)b/c^2 > 1$ so we can write $(1-\lambda)c^2/b = 1+\lambda'$, and then $Pr[Y_{c^2l} \geq (1+\lambda)c^2/bE[Y_{c^2l}]] \leq \exp(-\lambda'^2 * E[Y_{c^2l}]/3)$, and by choosing k large enough, l can be made large enough so that this can be made as small as we like. ■

Theorem A.4 (Theorem A.3 (Backpointers)) *If node A has a primary forward pointer to B (so A is the closest node to B with prefix $\alpha \circ j$), then with high probability, A is among the $k = O(\log n)$ closest nodes to B.*

Proof: We will show that the probability that A is not among the k closest nodes to B can be made arbitrarily small. Let $d = d(A, B)$ or the distance between A and B. Consider the ball around A of radius d. (Shown in Figure 11). Since B is the neighbor table of A, there is no node in this ball with a matching prefix. Further, notice that the ball around B containing k nodes does not contain A, so its radius must be less than d. Finally, consider the ball around A of radius $2d$. It completely contains the ball around B. We know, then, that the ball around A of radius d contains no nodes with prefix $\alpha \circ j$, but the ball around A of radius $2d$ contains k nodes of prefix α.

Let l be the number of nodes in the number of nodes in the smaller ball around B. We have two cases, first for large l, and second for small l.

case 1 $l \geq a \log n$. The probability that there is no matching node is $(1-p)^l$, where p is the probability a node has the prefix $\alpha \circ j$.
case 2 $l \leq a \log n$. Now, we consider the probability the larger ball has k α-nodes. Let S be the total number of nodes with prefix α. Then $E[S] = lbp$. Since S can be considered as the sum of boolean random variables, we can write that $Pr[S \geq (1+\lambda)E[S]] \leq \exp(-\lambda^2 E[S]/6)$, if we set $\lambda = k/E[S] - 1$, then we get that $Pr[S \geq k] \leq \exp(-(k/E[S]-1)^2 E[S]/6)$. Now, let $k = i \log n$ for i such that $i > 2a$. Then this is less than or equal to $\exp(-(i \log n/E[S])^2 E[S]/6) \leq \exp(-i \log n)$. ■

A.4 Acknowledged Multicast

Theorem A.5 (Theorem 5) *When a multicast recipient sends the acknowledgment, all the nodes with the given prefix have been reached.*

Proof: Suppose that node A receives a multicast message for prefix α and A is the only node when prefix A. Then the claim is trivially true.

Now, we assume the claim holds for a prefix α of length i, and we will prove it then holds for a prefix α of length $i-1$. Suppose node A receives a multicast message for a prefix of length α. The A sends the multicast to a node with every possible one-digit extension of α (i.e., $\alpha \circ j$ for all $j \in [0, b-1]$. Once A receives all those acknowledgments, all nodes with prefix α have been reached. Since A waits for these acknowledgments before sending its own, when A sends its acknowledgment, all nodes of prefix α have been reached. ■

The following theorem does not work unless the multicast prefix is defined as specified in the insertion algorithm. This does not mean that multiple insertions fall apart; it just means that the theorem statements become complicated and unwieldy, though the same ideas carry through.

Theorem A.6 (Theorem 6) *Suppose A and B both insert. If there is a node that receives A's multicasts before B's, and some other node that receives B's multicast before A's, then A will receive B's multicast if A should receive B's multicast.*

Proof: There are three cases:
case 1: A multicasted to a shorter prefix than B. This case is easy, since A does not need to receive B's message. **case 2:** A and B are multicasting to the same prefix. Suppose X has already received A's multicast. It now gets B's multicast. If A has not already be contacted, then the first-level hole pattern shows a hole where A should be, and X forwards the message to B. **case 3:** A multicasted to a longer prefix than B. This is very like the previous case. At some point in the multicast tree for B, there was a node that either sent the message to A, or decided that it could not send the message to any any node with that prefix of A. The subtree formed under that is exactly the set of nodes reached by A's, so we know X is in that subtree. That means that the holepattern X gets with B's multicast shows a hole for A, so X can forward the message to A. ■

When more three or more nodes are inserted at once, the situation becomes more complicated, because the newly inserted nodes are more than leaves in the multicast. In that case, they must maintain a list of received multicasts and forward those multicasts, if appropriate, as they learn about new nodes in the network.