Distributed Object 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

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 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 efficiently achieved for certain network topologies. To do this, we present a new distributed algorithm that can solve the nearest-neighbor problem for these networks. We describe our solution in the context of Tapestry, an overlay network infrastructure that employs techniques proposed by Plaxton, Rajaraman, and Richa [14].

Categories and Subject Descriptors

E.1 [**Data Structures**]: Distributed data structures; E.1 [**Data Structures**]: Graphs and Networks; F.2.2 [**Nonnumerical Algorithms and Problems**]: Routing and layout; C.2.4 [**Distributed systems**]: Distributed applications

General Terms

Algorithms,Theory

Keywords

Tapestry, networking, overlay, locality, peer-to-peer, distributed object location, DOLR, distributed hash table, DHT, nearest neighbor

1. INTRODUCTION

In today's chaotic network, data and services are mobile and replicated widely for availability, durability, and locality[1]. This has lead to a renewed interest in techniques for routing queries to objects using names that are independent of their locations. The notion of *routing* is that queries are forwarded from node to node until they reach their destinations. The *location-independent routing* problem has spawned a host of proposals, many of them in the context of data sharing infrastructures such as OceanStore [11], FarSite [3], CFS [9] and PAST [17]. 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.

Properties that we would like from a location-independent routing infrastructure include:

1. *Deterministic Location*: Objects should be located if they exist anywhere in the network.
2. *Routing Locality*: Routes should have low *stretch*[2], not just a small number of application-level hops. Sending queries to the nearest copy across the shortest path possible is the ideal.
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 system must adapt to arriving and departing nodes while maintaining the above properties.

Although clearly desirable, the first property is not guaranteed by existing peer-to-peer systems such as Gnutella [12] and FreeNet [5].

A simple object location and routing scheme would employ a centralized directory of object locations. Servers would *publish* the existence of objects by inserting entries into the directory. Clients would send *queries* to the directory, which forwards them to their destinations. 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 potentially distant directory server. The average routing latency of this technique is proportional to the average diameter of the network – independent of the actual distance to the object. Worse, it is neither fault tolerant nor scalable, since the directory becomes a single point of both failure and contention.

Several recent proposals, Chord [18], CAN [15] and Pastry [16], address the load aspect of this problem by distributing the directory information over a large number of nodes. In particular, they 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, while these approaches use a number of overlay hops that is polylogarthmic, the actual network latencies incurred by queries can be significantly more than those incurred by finding the object in the centralized directory.

*This research supported by NSF career award #ANI-9985250, NFS ITR award #CCR-0085899, NSF Graduate Research Fellowship, and UC MICRO award #00-049.

[1] By locality we mean the ability to exploit local resources over remote ones whenever possible [21].

[2] Stretch is the ratio between the distance traveled by a query to an object and the minimal distance from the query origin to the object.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPAA'02, August 10-13, 2002, Winnipeg, Manitoba, Canada.
Copyright 2002 ACM 1-58113-529-7/02/0008 ...$5.00.

RIGHTSLINK

Scheme	Insert	Space	Stretch, Metric	Hops	Balanced?
CHORD [18]	$O(\log^2 n)$	$O(n \log n)$	-	$O(\log n)$	yes
CAN [15]	$O(r)$	nr	-	$rn^{1/r}$	yes
Pastry [16]	$O(\log^2 n)$	$O(n \log n)$	-	$O(n \log n)$	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 [14]	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. In some cases, various messages can be sent in parallel; we did not allow for this optimization in stating the bounds in this table.

An alternative solution is to broadcast an object's location to every node in the network. This allows clients to easily find the nearest copy of the object, but requires a large amount of resources to publish and maintain location information, including both network bandwidth and storage. Furthermore, it requires full knowledge of the participants of the network. In a dynamic network, maintaining a list of participants is a significant problem in its own right.

We describe our results in the context of the Tapestry overlay routing and location infrastructure [22]. Tapestry uses as a starting point the distributed data structure of Plaxton, Rajaraman, and Richa [14], which we will refer to as the PRR scheme. Their proposal yields routing locality with balanced storage and computational load. However, it does not provide dynamic maintenance of membership. The original statement of the algorithm required a static set of participating nodes as well as significant work to preprocess this set to generate a routing infrastructure. Additionally, the PRR scheme was unable to adapt to changes such as node failures. This paper extends their algorithms to a dynamic network.

1.1 Related Work

Schemes that exhibit routing locality include Plaxton, Rajaraman, and Richa (PRR) [14] 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. Recall that stretch is the ratio between the actual latency or distance to an object and the shortest distance. The PRR scheme finds objects with constant stretch for a specific class of network topologies while ensuring that no node has too many directory entries. Awerbuch et al. route with stretch 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, 7, 13, 20] whose techniques are closely related to those in this paper. See [8] for a survey. A recent and closely related paper is that of Thorup and Zwick, who showed that a sampling based scheme similar to that of PRR could be used to find small stretch routing tables and/or answer approximate distance queries in arbitrary metric spaces[3].

Most of the recent work on peer-to-peer networks ignores stretch. Chord [18] constructs a distributed lookup service using a routing table of logarithmic size. 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 [15] places objects into a virtual, high-dimensional space. Queries are routed along axes in this virtual space until they reach their destination. Pastry [16] is loosely based on the PRR scheme, routing queries via successive resolution of digits in a high-dimensional name space. While its overlay construction leverages network proximity metrics, it does not provide the same stretch as the PRR scheme in object location. 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. In addition, Chord and CAN have run-time heuristics to reduce object location cost, so they may perform well in practice. Further, all of these can support the introduction and removal of nodes.

Recent peer-to-peer systems can locate objects in a dynamic network. Gnutella [12] 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[4]. Neither of these techniques are guaranteed to find objects.

Table 1 summarizes related work alongside our contributions. Systems with no entry in the "Stretch, Metric" column do not consider stretch at all; those with "special" assume the metric space has a certain low-expansion property described in Section 3.

1.2 Results

Our goals are not only to derive the best possible asymptotic results, but also to analyze the simple schemes that are the basis of the PRR and the Tapestry algorithms. This paper includes three main results:

- We present a simplification of the PRR scheme for object location. We cannot prove that this object location scheme meets the same bounds on stretch as the PRR scheme; however, it appears to perform well in practice.
- We extend this scheme (as well as the PRR approach) to deal with a changing participant set. We allow nodes to arrive and depart while maintaining the ability to locate existing objects and publish new objects. This works for a slightly broader class of metric spaces than assumed by PRR.
- We observe that a static version of the PRR scheme can be used for general metric spaces (*i.e.* spaces that do not meet the conditions assumed by PRR) to get results similar to those of Awerbuch and Peleg [1].

[3] A network topology gives a metric space.

[4] This is a qualitative statement at this time.

RIGHTSLINK

Table 1 gives a summary of some of the previous results along with ours. Note that the result for general metrics can be improved using results of Thorup and Zwick [19] to use only $O(n \log n)$ space.

Techniques: The crux of our method for inserting nodes into the network lies in an algorithm for maintaining nearest neighbors in a restricted metric space. Our approach follows that of Karger and Ruhl [10], who give a sequential algorithm for answering nearest neighbor queries in a similarly restricted metric space[5].

Karger and Ruhl describe a data structure using a random permutation to help maintain a random sampling. This approach is dynamic and reminiscent of the Chord network infrastructure. Our data structure uses random names to get a random sampling.

We also prove that an alternate scheme by Plaxton, Rajaraman, and Richa (called PRR v.0 in Table 1) 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.

The remainder of this paper is divided as follows: Section 2 describes the details of Tapestry, highlighting differences with the PRR scheme and introducing concepts and terminology for the remainder of the paper. Section 3 describes how to solve the incremental nearest neighbor problem. Section 4 explains how this is used as part of inserting a node. Section 5 discusses deletion. Section 6 gives a simple proof that PRR v.0 scheme has polylogarithmic stretch for general metric spaces. We conclude in Section 7.

2. THE TAPESTRY INFRASTRUCTURE

Tapestry [22] is the wide-area location and routing infrastructure of OceanStore [11]. Tapestry assumes that nodes and objects 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 object 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) [14]. 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.

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

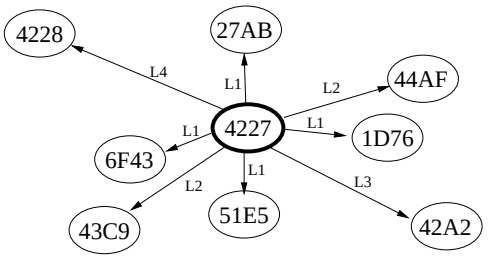

Figure 1: *Tapestry Routing Mesh.* Each node is linked to other nodes via *neighbor links*, shown as solid arrows with labels. Labels denote which digit is resolved during link traversal. Here, node `4227` has an L1 link to `27AB`, resolving the first digit, an L2 link to `44AF`, resolving the second digit, etc. Using the notation of Section 2.1, 42A2 is a $(\texttt{42}, \texttt{A})$ neighbor of `4227`.

Neighbors for node A are grouped into *neighbor sets*. For each prefix β of A's ID and each symbol $j \in [0, b-1]$, the neighbor set $\mathcal{N}^A_{\beta,j}$ contains Tapestry nodes whose node-IDs share the prefix $\beta \circ j$. We will refer to these as (β, j) neighbors of A or simply (β, j) nodes. For each j and β, the closest node in $\mathcal{N}^A_{\beta,j}$ is called the primary neighbor, and the other neighbors are called secondary neighbors. When context is obvious, we will drop the superscript A. Let $l = |\beta| + 1$. Then, the collection of b sets, $\mathcal{N}^A_{\beta,j}$, 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 $K \geq 1$: $|\mathcal{N}^A_{\beta,j}| \leq K$. Further, $|\mathcal{N}^A_{\beta,j}| < K$ implies $\mathcal{N}^A_{\beta,j}$ contains all (β, j) nodes. This gives us the following:

PROPERTY 1 (CONSISTENCY). *If $\mathcal{N}^A_{\beta,j}=\phi$, for any A, then there are no (β, j) nodes in the system. We refer to this as a "hole" in A'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 A_0 and destination node be B, with a node-ID equal to $\beta \equiv j_1 \circ j_2 \ldots j_n$. Then routing proceeds by choosing a succession of nodes: $A_1 \in \mathcal{N}^{A_0}_{\phi,j_1}$ (first hop), $A_2 \in \mathcal{N}^{A_1}_{j_1,j_2}$ (second hop), $A_3 \in \mathcal{N}^{A_2}_{j_1 \circ j_2, j_3}$ (third hop), *etc.* We would like to emphasize

PROPERTY 2 (LOCALITY). *In both Tapestry and PRR, each $\mathcal{N}^A_{\beta,j}$ contains the closest (β, j) neighbors as determined by a given metric space. The closest neighbor with prefix $\beta \circ j$ is the* 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 A can find its nearest neighbor by choosing from the set $\bigcup_{j \in [0,b-1]} \mathcal{N}^A_{\phi,j}$. Section 3 will discuss how to maintain Property 2 in a dynamic network.

2.2 Routing to Objects with Low Stretch

Tapestry maps each object 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.

RIGHTSLINK

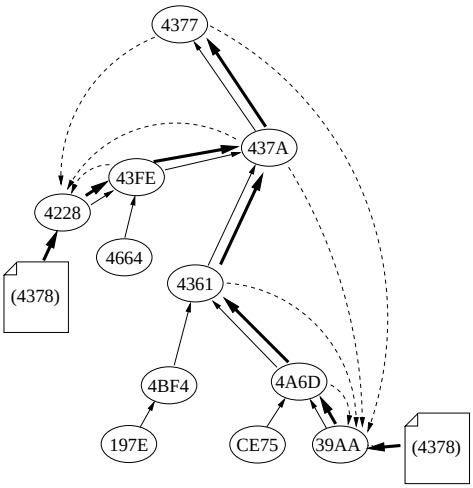

Figure 2: *Publication in Tapestry.* To publish object `4378`, server `39AA` sends publication request towards root, leaving a pointer at each hop. Server `4228` publishes its replica similarly. Since no `4378` node exists, object `4378` is rooted at node `4377`.

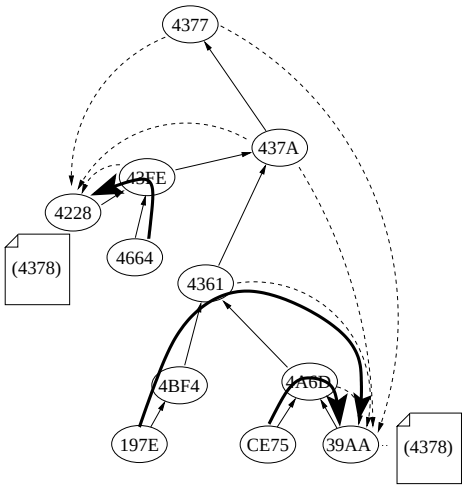

Figure 3: *Routing in Tapestry:* Three different location requests. For instance, to locate GUID `4378`, query source `197E` routes towards the root, checking for a pointer at each step. At node `4361`, it encounters a pointer to server `39AA`.

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 objects. In the simplest version of Tapestry, $|\mathcal{R}_\psi| = 1$. In this case, we can speak of *the root node* for a given node, ψ. For this to be sensible, we must have the following property:

PROPERTY 3 (UNIQUE ROOT SET). *The root set, $\mathcal{R}_\psi$, for object 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. To provide fault-tolerance, Tapestry assumes that pointers are *soft-state*, *i.e.* pointers expire and objects must be republished (that is, published again) at regular intervals. Republishing may be requested if something changes about the network.

Queries for object 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 to the current node (i.e. the node where the object pointer was found). At the beginning of the query, we select a root 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 be very likely to intersect the publishing path before reaching the root.

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. The following theorem shows an important property shared by PRR and Tapestry.

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. □

OBSERVATION 1. *(Fault Tolerance) If $|\mathcal{R}_\psi| > 1$ and the names in $\mathcal{R}_\psi$ are independent of one another, then we can retry object queries and tolerate 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 with a certain even-growth property: 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, [14] shows the average distance traveled in locating an object is *proportional* to the distance from that object, *i.e.* queries exhibit $O(1)$ stretch. Unfortunately, the constants in their proof make for an impractical system. Tapestry is easier to implement and seems to provide low stretch in practice [22].

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. 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 this process *surrogate routing*, since it involves routing toward ψ as if it were a node, then adapting when the process fails.

RIGHTSLINK

```
method AcquireNeighborTable (NewNodeName,NewNodeIP, PSurrogateName,PSurrogateIP)
1  α ← GreatestCommonPrefix(NewNodeName, PSurrogateName)
2  maxLevel ← Length(α)
3  list ← AcknowledgedMulticast [on PSurrogateIP](α, SendID(NewNodeIP, NewNodeName))
4  BuildTableFromList(list, maxLevel)
5  for  i = maxlevel - 1 to  0
6       list ← GetNextList(list, i)
7       BuildTableFromList(list, i)
   end AcquireNeighborTable

   method GetNextList (neighborlist, level)
1  nextList ← ∅
2  for n ∈ neighborlist
3      temp ← GetForwardAndBackPointers(n, level))
4      nextList ← KeepClosestk(temp ∪ nextList)
5  return  nextList
   end GetNextList
```

Figure 4: Building a Neighbor Table. A few words on notation: FUNCTION **[on destination] represents a call to run** FUNCTION **on destination, variables in italics are single-valued, and variables in bold are vectors.**

Roots reached in this way are considered *surrogate roots* of ψ.

In a dynamic network, maintenance of these additional pointers can be problematic, 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 routing can go no further (the only node left at and above the current level is the current node), that node is the root. This scheme is simpler than PRR 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.*

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. □

Surrogate routing in Tapestry may introduce additional hops over PRR; however, the number of additional hops is independent of n and in expectation is less than 2 [22]. Notice the following:

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.*

3. BUILDING NEIGHBOR TABLES

Building the neighbor table is the most complex and interesting part of the insertion process, so we discuss it first. We want to build the neighbor sets, $\mathcal{N}^{A}_{\beta,j}$ for a new node A. These sets must adhere to Properties 1 and 2. This amounts to solving the nearest neighbor problem for many different prefixes. We could simply use the method of Karger and Ruhl [10] many times, once for each prefix. The method we present below has lower network distance, though as many network hops as a straightforward use of Karger and Ruhl. Our method incurs no additional space over the PRR solution.

As in [14], we adopt the following network constraint. Let $\mathcal{B}_A(r)$ denote the ball of radius r around A; *i.e.*, all points within distance r of A, and $|\mathcal{B}_A(r)|$ denote the number of such points. We assume:

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

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 almost exactly that used by Karger and Ruhl [10]. We also assume the triangle inequality, that is $d(X,Y) \leq d(X,Z) + d(Z,Y)$ for any set of nodes X, Y, and Z. Finally, we will give bounds in terms of network latency or network hops ignoring local computation. None of the local computation is expensive, so this reasonable.

Figure 4 shows how to build neighbor tables. Suppose that the longest common prefix of the new node and any other node in the network is α. Then we start with a list of all nodes with prefix α. (We explain how to get this list in the next section.) Then we get similar lists for progressively smaller prefixes, until we have the closest k nodes matching the empty prefix.

To go from the level-$(i+1)$ list to the level-i list, we ask all the nodes on the level-$(i+1)$ list for all the level-i nodes they know about (we ask for both forward and backwards pointers). We then trim this list, keeping only the closest k nodes.

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 K (α_i, j) nodes (K is defined in Section 2.1). If $b > c^2$, then Theorem 3 says there is some $k = O(\log n)$ such that with high probability, we know there is one (or indeed, any constant) number of such nodes on the list for every j. (Note that PRR assume that $b \geq c^8$).

THEOREM 3. *If c is the expansion constant of the network and $b > c^2$ (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 closest level-$(i+1)$ nodes, we can find 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.

RIGHTSLINK

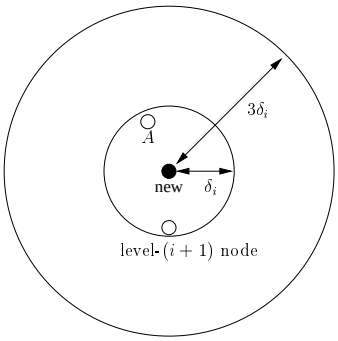

Figure 5: Figure for Theorem 3. If $3\delta_i$ is less than δ_{i+1}, then A must point to a node within d_{i+1}.

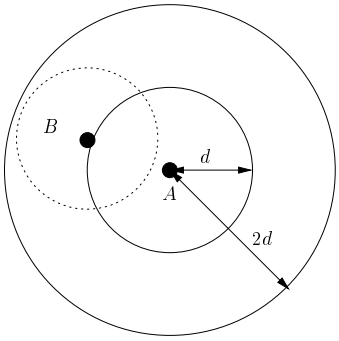

Figure 6: Figure for Theorem 4.The larger ball around A contains $O(\log n)$ nodes, while the smaller ball contains none.

We would like to show that any node A inside the ball must point to a level-$i+1$ node within δ_{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 node and A's parent is no more than $2\delta_i + \delta_i = 3\delta_i$. (See Figure 5.) This means that as long as $3\delta_i < \delta_{i+1}$, A must point to a node inside δ_{i+1}. Since we query all level-$(i+1)$ in δ_{i+1}, this means we will query A's parent, and so find A.

Now, we must show that $3\delta_i < \delta_{i+1}$ with high probability. This is the expected behavior; given a ball with k level-i nodes, doubling the radius twice gets no more than c^2k nodes, and so, no more than c^2k/b level-$(i+1)$ nodes. Since $c^2/b < 1$, this means that the quadrupled ball has less than k level-$(i+1)$ nodes, or equivalently, the ball containing k level-$(i+1)$ nodes is at least three (really, four) times the size of the ball with k level-i nodes. The following turns this informal expectation argument into a proof.

Choose λ such that $(1-\lambda)b/c^2 > 1$. This is possible since we know that $b/c^2 > 1$. Now, let l be $(1-\lambda)^{-1}kb^i$. This is the required number of nodes such that one expects $(1-\lambda)^{-1}k$ of the nodes to be level-i nodes. Now let l_{real} be the total volume of the ball containing k level-i nodes. We consider two cases.

Case 1: $l_{\text{real}} \geq l$. We argue that with high probability, this does not happen. Let X_m be a random variable representing the number of level-i nodes in m nodes. Then we wish to bound $Pr[X_{l_{\text{real}}} \leq k]$. But $Pr[X_{l_{\text{real}}} \leq k] \leq Pr[X_l \leq k]$, since $l \leq l_{\text{real}}$. But $Pr[X_l \leq k] = Pr[X_l \leq (1-\lambda)E[X_l]]$. Using a Chernoff bound, this is less than $\exp(-\lambda^2 E[X_l]/2) \leq \exp(-\lambda^2 k/2)$ and if $k = a\log n$ with a sufficiently large, this can be made as small as desired.

Case 2: $l_{\text{real}} \leq l$. 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 smaller that $3\delta_i$), so the ball of radius $4\delta_i$ must also contain at least k level-$(i+1)$ nodes.

However, we know the volume (that is, the number of nodes) of this ball is less than $c^2 l_{\text{real}}$ by Equation 1. Let Y_m be the number of $i+1$ nodes in m trials. We wish to bound $Pr[Y_{c^2 l_{\text{real}}} \geq k]$. But this $Pr[Y_{c^2 l_{\text{real}}} \geq k] \leq Pr[Y_{c^2 l} \geq k]$, and this is the same as $Pr[Y_{c^2 l} \geq (1-\lambda)b/c^2 E[Y_{c^2 l}]]$, since $E[Y_{c^2 l}] = c^2/bE[Y_l] = c^2/b(1-\lambda)^{-1}k$. We know that $(1-\lambda)b/c^2 > 1$ so we can write $(1-\lambda)c^2/b = 1+\lambda'$, and then $Pr[Y_{c^2 l} \geq (1+\lambda)c^2/bE[Y_{c^2 l}]] \leq \exp(-\lambda'^2 * E[Y_{c^2 l}]/3)$, and by choosing k large enough, l can be made large enough so that this can be made as small as we like.

So, by choosing k large enough, we can guarantee that neither case happens with high probability. □

The new node also induces changes on other nodes' neighbor tables. Theorem 4 shows that there is a $k' = O(\log n)$ such that any node that needs to update its level-i link is one of the closest k' nodes for level-i with high probability. If we choose $k \geq k'$ we know that with high probability, such a node will be contacted during the insertion, and so it will add the new node to its neighbor table. We assume 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 B is the primary (α, j) neighbor of A (so B is the closest node to A with prefix $\alpha \circ j$), then with high probability, A is among the $k = O(\log n)$ closest α-nodes to B.*

PROOF. We will show that the probability that A is not among the k closest α-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 6). Since B is the neighbor table of A, there is no (α, j) node in this ball. Further, notice that the ball around B containing k α-nodes does not contain A (or else the proof is done), so its radius must be less than d. Finally, consider the ball around A of radius $2d$. It completely contains the ball around B. Thus, the ball around A of radius d contains no (α, j) nodes, but the ball around A of radius $2d$ contains k nodes of prefix α.

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

Case 1: $l \geq (a/p)\log n$, where p is the probability a node is (α, j)-node. The probability that there is no (α, j) node is less than $(1-p)^l \leq \exp(-a\log n)$. This can be made as small as necessary by choosing a large enough.

Case 2: $l \leq (a/p)\log n$. Now, consider the probability the larger ball has k or more α-nodes. Let S be a random variable representing the number of α nodes in the larger ball. Since that ball has volume l, $E[S] = lbp$ (b is the logarithm base).

Since S is the sum of boolean random variables, we can write that $Pr[S \geq (1+\lambda)E[S]] \leq \exp(-\lambda^2 E[S]/6)$. Consequently, 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 $i > 2a$. Notice that $(k/E[S]-1) \geq i\log n/E[S]$. Substituting and simplifying, we get that $Pr[S \geq k] \leq \exp(-i/6\log n)$.

RIGHTSLINK

```
method INSERT (gatewayIP, NewNodeIP, NewNodeName )
1  (PSurrogateIP, PSurrogateName) ← ACQUIREPRIMARYSURROGATE (gatewayIP, NewNodeName)
2  α ← GREATESTCOMMONPREFIX(NewNodeName, PSurrogateName)
3  GETPRELIMNEIGHBORTABLE [on PSurrogateIP]()
4  ACKNOWLEDGEDMULTICAST [on PSurrogateIP](α, LINKANDXFERROOT[NewNodeIP, NewNodeName])
5  ACQUIRENEIGHBORTABLE (NewNodeName,NewNodeIP,PSurrogateIP, PSurrogateIP)
end INSERT
```

Figure 7: 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(α,FUNCTION)
1  if NOTONLYNODEWITHPREFIX(α)
2     for i = 0 to b − 1
3        neighbor ← GETMATCHINGNEIGHBOR(α ∘ i)
4        if neighbor exists
5           S ← ACKNOWLEDGEDMULTICAST [on GETIP(neighbor)] (α ∘ i, FUNCTION )
6  else
7     apply FUNCTION
8  wait S
9  SENDACKNOWLEDGEMENT()
end ACKNOWLEDGEDMULTICAST
```

Figure 8: Acknowledged Multicast

To bound the probability of either case, choose a so that the probability of case 1 is made small, and then choose i to make the probability of case 2 small. □

Since each node has an expected constant number of pointers per level, the expected time of this algorithm is $O(k) = O(\log n)$ per level or $O(\log^2 n)$ overall. (We are concerned with network traffic and distance and hence ignore the cost of sorting those k distances.)

The number of backpointers is less than $O(\log n)$ per level per node with high probability, so we get a total time of $O(\log^3 n)$ with high probability. But this analysis can be tightened. 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, again 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. (We believe that the analysis can be further tightened to show that with high probability, only $O(\log n)$ nodes are touched, total.)

Further, 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) that $q = O(\log n)$. The total network latency is bounded by:

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

Since the δ_i are geometrically decreasing, they sum to $O(d)$, where d is the network diameter, so the total latency for building neighbor tables is $O(qd) = O(d \log n)$.

4. NODE INSERTION

Next, we describe the overall insertion algorithm, using the nearest neighbor algorithm as a subroutine. We would like the results of 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 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 hold 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 7 shows the basic insertion algorithm. First, the new node contacts its surrogate; that is, the node with the ID closest to its own. Then it gets a copy of the surrogate's neighbor table. These first two steps could be combined, if desired. Next 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 in Section 4.1) to do this. As a final step, we build the neighbor tables, as described in Section 3. To reduce the number of multicasts, we can use the multicast in step 3 of the insertion algorithm to get the first **list** of the nearest neighbor algorithm. Finally, notice that once the multicast is finished, the node is fully functional, though its neighbor table may be far from optimal.

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 failure cases, one of correctness, where not fixing the problem means that the network may fail to locate an object, and one of performance, where not fixing the problem may increase object location latency.

The function LINKANDXFERROOT from Figure 7 takes care of correctness by transferring object pointers that should be rooted at the new node and deleting pointers that should no longer be on the current node. If we do not move the object pointers, then objects may become unreachable. Performance optimization involves redistributing pointers and will be discussed in Section 4.2.

4.1 Acknowledged Multicast

To contact all nodes with a given prefix we introduce an algorithm called *Acknowledged Multicast*, shown in Figure 8. This algorithm is initiated by the arrival of a multicast message at some node.

A multicast message consists of a prefix α and a function to ap-

RIGHTSLINK

```
method OPTIMIZEOBJECTPTRS (sender,changedNode,objPtr,level)
1       oldsender ← GETOLDSENDER(objPtr)
2       if oldsender ≠ null and oldsender ≠ sender
3         OPTIMIZEOBJECTPTRS [on NEXTHOP(objPtr,level)](self ,changedNode,objPtr,level + 1)
4         if oldsender ≠ changedNode
5           DELETEPOINTERSBACKWARD [on oldsender](objPtr, changedNode,level - 1)
end OPTIMIZEOBJECTPTRS
method DELETEPOINTERSBACKWARD (changedNode,objPtr,level)
1       oldsender ← GetOldSender(objPtr)
2       DELETE(objPtr)
3       if oldsender ≠ changedNode
4         DELETEPOINTERSBACKWARD [on oldsender](objPtr, changedNode,level - 1)
end DELETEPOINTERSBACKWARD
```

Figure 9: OptimizeObjectPtrs and its helper function

```
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 10: Misrouting and route correction to maintain object availability

ply. To be a valid multicast message, 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 (α, j) node if such a node exists. One of these extensions will be the node itself, so a node may receive multicast messages from itself at potentially many different levels. We know by Property 1 that if an (α, j) node exists, then every α-node knows at least one such node. Each of these nodes then continues the multicast. When a node cannot forward the message further, it applies the function.

Because we need to know when the algorithm is finished, we require each recipient to send an acknowledgment to its parent after receiving acknowledgments from its children. If a node has no children, it sends the acknowledgment immediately. When the initiating node 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 with prefix α sends acknowledgment, all the nodes with prefix α have been reached.*

PROOF. This is a proof by induction on the length of α. In the base case, suppose node A receives a multicast message for prefix α and A is the only node with prefix A. The claim is trivially true.

Now, assume the claim holds for a prefix α of length i. We will prove it holds for a prefix α of length $i-1$. Suppose node A receives a multicast for a prefix of length α. Then A forwards the multicast to one node with each 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, all nodes of prefix α have been reached when A sends its acknowledgment. □

These messages form a tree. If you collapse the messages sent by a node to itself, the result is in fact a spanning tree. This means that 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 $O(k)$ 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 $O(dk)$. Note that there is a variant of this algorithm that does not require maintaining state at the all the participating nodes, but this is beyond our scope.

4.2 Redistributing Object Pointers

Recall that object publish their location by placing pointers to themselves along the path from the server to the root. From time to time, we re-establish these pointers in an operation called republish. This section describes a special version of republish that maintains Property 4. This function is used to rearrange the object pointers any time the routing mesh changes the expected path to the root node for some object (*e.g.*when a node's primary neighbor is replaced by a closer node). This adjustment is not necessary for correctness, but does improve performance of object location.

If the node uses an ordinary republish (simply sending the message towards the root), it could leave object pointers dangling until the next timeout. For example, if the disappearance of node A changes the path from an object to its root node so that the path skips node B, then node B will still be left with a pointer to the object. Further, this type of republish may do extra work updating pointers that have not changed.

Instead, the node whose forward routes changed sends the object pointer up the new path. The new path and the old path will converge at some node, where a delete message is sent back down the old path, removing outdated pointers. This requires maintaining a last-hop pointer for each object pointer. See Figure 9.

Notice, however, that Property 4 is not critical to the functioning of the system. If a node should use OPTIMIZEOBJECTPTRS but does not, then performance may suffer, but objects will still be available. Further, timeouts and regular republishes will eventually ensure that the object pointers are on the correct nodes.

4.3 Keeping Objects Available

While a node is inserting itself, object requests that would go to the new node after insertion may either go to the new node or to

RIGHTSLINK

a pre-insertion destination. Figure 10 shows how to keep objects available during this process: if either node receives a request for an object it does not have, it forwards the request to the other node.

If an inserting node receives a request for an object it does not have, it sends the 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 at level $i+1$ to one of the surrogate nodes. The surrogate then routes the message as it would have if the new node had not yet entered the network.

If a pre-insertion root receives a request for an object pointer that has already been moved to the new node, it should forward the request to the new node. But we want to do this in such a way that the surrogate does not need to keep 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: 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 old root 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 one of the two nodes is guaranteed to have the pointer. No matter which node receives the request, before or after the transfer of pointers, the node servicing the request either has the information to satisfy the query or else it forwards the query to the other node, which can satisfy it using local information.

As a final point, 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. This allows the system to detect and prevent loops, and since the number of hops is small, this is not an unreasonable overhead.

4.4 Network Traffic

Without 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. Finding the surrogate is no more costly than searching for an object pointer, and [14] argues that finding an object pointer requires $O(d)$ messages. Multicast takes time $O(kd)$ where k is the number of nodes reached. But k will be small and constant relative to n. Finally, building the neighbor tables takes $O(\log^2 n)$ 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 gives a total traffic of $O(md \log n)$ for object pointer relocation.

4.5 Simultaneous Insertion

In a wide-area network, insertions may 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 more distant (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. Local optimization mitigates this problem. If an exact answer is desired, we can rerun the neighbor table building algorithm after a random amount of time.

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 could be solved by reinserting the node, but before the reinsertion occurs, objects may be unavailable. This is a serious problem.

We start with a definition:

DEFINITION 1. *Assume that we start with a consistent Tapestry network. A* core node *is a node that is completely integrated in this network, i.e. it has no holes in its neighbor table that can be filled by other nodes in the network, and it cannot fill holes in the neighbor tables of nodes in the network.*

Together, the core nodes all satisfy Property 1. By this definition, a node could be a core node without meeting locality Property 2. The goal of this section is prove that when a node finishes its multicast, it becomes a core node, and that all the nodes that were core nodes before its multicast finished remain core nodes.

Two operations are *simultaneous* if there is a point in time when both operations are ongoing. This straight-forward definition is important from a systems standpoint. It is a bit imprecise, however, since two multicasts could be simultaneous and yet be indistinguishable from sequential multicasts without the use of a global clock. We would like to distinguish between cases where all core nodes see evidence consistent with a sequential ordering and cases where there can be no such agreement. To this end, we say that two multicasts *conflict* if there are two nodes that receive the multicasts in different orders.

THEOREM 6. *Suppose A inserts. When it is done with its multicast, A's table has no holes that can be filled by core nodes. Further, there are no core nodes with holes that A can fill. These statements are true even when other insertions proceed simultaneously.*

We start by proving a series of lemmas. Our first Lemma is simple, but important. Lemma 1 states that simultaneously inserting nodes cannot interfere with one another's access to core nodes.

LEMMA 1. *Nodes in S, the set of core nodes, can be reached by a given multicast even in the presence of ongoing or completed insertions of other nodes.*

PROOF. Proof by contradiction. Theorem 5 says that all core nodes can reach one another. Suppose there is a multicast that misses node $X \in S$. Let B be the node that should have sent the multicast towards X but did not. Further, suppose that the prefix B received with the multicast was α. If B did not send the multicast towards X (that is, send it to (α, j) where $\alpha \circ j$ is a prefix of X's ID), it must have been because it did not have a (α, j) node in its table. But this is not possible:

Case 1: B has not yet finished its multicast. Since B is supposed to send the multicast to X, we know that B and X must share prefix α. Further, since we know that X was in the network before B began its multicast, B's multicast set would have been the nodes with prefix α. But this means that B would only have filled (α, j) entries, so it could not possibly have been contacted with a prefix smaller than $\alpha \circ j$. Contradiction.

Case 2: B has finished its multicast and is a core node by Theorem 6, and since X is an (α, j) node, B must have such a node in its table. Contradiction.

Although Theorem 6 uses Lemma 1, Case 2 is not circular: Node B was inserted *before* the point in time that we use it here. □

RIGHTSLINK

```
method AcknowledgedMulticast(α, Function, holebeingfilled, watchlist, NewNodeIP)
1  watchlist ← CheckForNodesAndSend(watchlist, NewNodeIP)
2  if NotOnlyNodeWithPrefix(α)
3     for i = 0 to b − 1
4        neighbor ← GetMatchingNeighbor(α ∘ i)
5        if neighbor exists
6           S ← AcknowledgedMulticast [on GetIP(neighbor)]( α ∘ i, Function, holebeingfilled, watchlist, NewNodeIP )
7  else
8     apply Function
9     S ← MulticastToFilledHole(holebeingfilled, Function, watchlist, NewNodeIP)
10 wait S
11 SendAcknowledgement()
   end AcknowledgedMulticast
```

Figure 11: Acknowledged multicast with the watch list

Now we must deal with the case where two insertions conflict. We first introduce the notion of a locked pointer. An (α, j) pointer to node A stored at node X is *locked* when there are nodes whose multicasts through (α, j) have arrived at X but have not been acknowledged.

When a multicast for a new node filling an (α, j) slot arrives at some node X, X puts the new node in the table as a locked pointer and sends the multicast to one unlocked pointer and all locked pointers. When X receives acknowledgments from all recipients, X unlocks the pointer. Finally, X must keep at least one unlocked pointer and all locked pointers. If this is done, then X will reach all (α, j) nodes it knows about without having to store them all. Intuitively, the unlocked pointer can reach all other unlocked pointers so unlocked pointers are all equivalent, while the locked pointers are not well-enough connected to be reachable via multicast.

LEMMA 2. *A multicast through an unlocked (α, j) pointer at node X reaches all other nodes that have or had unlocked (α, j) pointers at node X.*

The proof is similar to other multicast arguments. Ideally, each multicast will see the other as completed. To enforce this condition, if any node gets a multicast from A, and notices that the hole for A is already filled, it contacts all nodes it has seen that fill that hole. As above, it contacts one unlocked pointer and all the locked pointers.

Next, we deal with the case when A and B fill the same hole.

LEMMA 3. *Suppose A and B fill the same hole. Then with the modification described above, if A's multicast conflicts with B's, A will get B's multicast message before B's multicast is finished.*

PROOF. Let X be a node that gets A's multicast before B's. Then when X gets B's multicast, it forwards it on to A, since they fill the same hole. Finally, since X does not a send an acknowledgment until A returns an acknowledgment of B's multicast, A has been informed by the the time B's multicast finishes. We then apply this same argument with the roles of A and B reversed. □

This does not solve every problem. Consider when the $\alpha \circ i$ hole and the $\alpha \circ j$ hole are both being filled by two different nodes (with $i \neq j$). Then the $\alpha \circ i$ node may not get the $\alpha \circ j$ multicast and vice versa, even though their multicast sets are the same.

So, we further modify the multicast. The starting node sends down a "watch list" of prefixes for which it knows no matching node. This can be represented as a bit vector. When the inserting node sends this to the surrogate, it is a zero for every entry in the neighbor table. Each receiving node checks the watch list to see if it can fill in any blank on the list. If it can, it sends the relevant node to the originator of the multicast, marks the entry as found, and continues the multicast. From this description, it may sound as if we are sending a lot of information; in fact, we will be sending very little, since most of the lower levels of the table will be filled by the surrogate in the first step, and most of the upper levels of the table will be zero. In the normal case, we send only a few levels of the neighbor table, and each level is sixteen bits. This new version is shown in Figure 11. Using this new multicast, we get Lemma 4.

LEMMA 4. *Let A be an α node, and let B be an (α, j) node. Then if the core α nodes get multicast messages from both A and B, the (α, j) slot on A will not be a hole. (The core nodes are those that have finished their multicasts when the latter of A and B start its multicast.)*

PROOF. There are two cases.

Case 1: One α node, X, gets B's multicast first and then A's. In this case, when A's multicast arrives on X, X checks A's watch list, and if the watch list has an (α, j) hole B can fill, X has that hole filled and so will be able to notify A that it too can fill that hole. If there is no hole in the watch list, then A has already found such a node.

Case 2: All core α nodes gets A's multicast first. This means that A gets the multicast about B.

This completes the proof. □

Finally, we put everything together and prove Theorem 6.

PROOF. Consider a node B, and let α be the longest shared prefix between A and B.

Case 1: If A and B fill different holes on the same level (i.e., A fills an (α, i) hole and B fills (α, j) hole for $i \neq j$) , then they multicast to the same prefix α. By Lemma 1, we know these nodes are reached, and we can apply Lemma 4, once with A in the theorem being A of the lemma, and once with A of this theorem as B in the lemma.

Case 2: If A and B fill different holes on different levels, then we know that there are core α nodes in the network, and by Lemma 1 we know these nodes are reached. Given that, we again apply Lemma 4.

RIGHTSLINK

```
method DELETESELF ()
1       for pointer in { backpointers }
2           level = GETLEVEL(pointer)
3           LEAVINGNETWORK[on GETIP(pointer)](selfID, level, GETNEAREST(pointer, level))

4       for pointer in {neighbors ∪ backpointers }
5           REMOVELINK[on GETIP(pointer)](selfID)
end DELETESELF
```

Figure 12: Voluntary Delete

Case 3: If A and B fill the same hole on the same level, then there could be no core node with prefix α so the preceding arguments fail. In this case, we rely on Lemma 3, which says that if the two multicasts are not serialized, each will find out about the other before their multicasts complete.

This completes the proof. □

This algorithm can result in multiple multicasts to a new node. However, this is uncommon and it is rare that the new node will be anything other than a leaf in the tree (i.e. the new node will not forward the multicast). Further, the new node can easily suppress duplicate multicast messages.

5. DELETE

In this section, we discuss what happens when 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 an optimistic situation that permits fixing of neighbor links and object pointers. An *involuntary* delete occurs when a node ceases to function without warning.

5.1 Voluntary Delete

When node A volunteers to leave the network, it removes itself in a way that permits seamless object location. It starts by following its backpointers to notify corresponding nodes that it intends to leave the network. On each such node (say N), links to A are marked as "deleting" and replaced with other node links. Then, object pointers on N are republished as if node A did not exist. While pointers to A are marked as "deleting", queries continue to be routed to A, but publish operations are routed around A. See Figure 12.

Asking a node to delete its links to A could leave this node with an incorrect hole in its routing table (breaking Property 1). The more secondary pointers it has, the less likely this is to be true. However, to help avoid this problem, we send it suggested replacements. Given this information, the node can maintain an approximately correct neighbor table. It may wish to run the nearest-neighbor algorithm periodically to tune the neighbor table.

Once node A has contacted all of its backpointers, objects that were rooted at A are now reachable through new surrogates. As a final step, it simply goes through all backpointers and forward pointers and tells those nodes that it is leaving. This removes all links to A (including those marked as "deleting").

5.2 Involuntary Delete

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 dictate state changes to any other node. However, in the process of fixing its state, it may hint to other nodes that their state may be out of date. Deletion can be detected by soft-state beacons [22] or when a node sends a message to a defunct node and does not get a response.

When node B detects a faulty node, it should first remove the node from its neighbor table, then find a suitable replacement. B can find a replacement using a simple local search algorithm; that is, asking its remaining neighbors for their nearest matching nodes. This is not guaranteed to give the closest replacement node. Alternatively, the nearest neighbor algorithm can be repeated. If this produces a hole in the table, B will have to do additional work, described below, to ensure Property 1 is maintained. In any case, it should also use OPTIMIZEOBJECTPOINTERS on all object pointers that would have gone through the deleted node.

To ensure Property 1, if deleting the node leaves a hole in the routing table, B contacts the deleted node's surrogate. The surrogate either replies with a replacement node, or performs a multicast to all nodes sharing the same prefix as B and the dead node. If none of those nodes knows of a node to fill the hole, then B can safely assume the hole cannot be filled, and the surrogate informs all the nodes touched in the multicast. Likewise, if the surrogate does find a node to fill the hole, it informs these nodes of the replacement. During the multicast, all nodes republish any objects that would route through the departed node. Note that choosing the surrogate is arbitrary; any node could run this multicast, and we choose the surrogate node to ensure agreement on the multicast source.

Unfortunately, this does not maintain object availability. Objects rooted at the deleted node may become unavailable until a republish arrives at the node's surrogate. In fact, a network partition may result in an inconsistent deletion; we do not address this here.

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 polylogarithmic 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 [14] without load balancing, and is similar to the scheme of Thorup and Zwick [19].

Let $S_{i,j}$ be a sample of the metric space such that each node is chosen with probability $2^i/n$, and let $i \in [1, \log n]$ and $j \in [0, c\log n]$. Pick a single node at random to be in $S_{0,0}$. 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 objects located at nodes which point to it.

Suppose node X wants to find object Y. Starting with $i = \log n$, X asks (for all j in parallel) its representative in the set $S_{i,j}$ if it knows of Y. If one of them does, it returns the pointer to Y. If this fails, it tries $S_{i-1,j}$ for all j. Recall that there is one node in $S_{0,0}$, so this will always find the object, if it exists.

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 $O(\log^2 n)$.*

RIGHTSLINK

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 $|\mathcal{B}_X(r_k) \cap \mathcal{B}_Y(r_k)| \geq \frac{1}{2} |\mathcal{B}_X(r) \cup \mathcal{B}_Y(r_k)|$ 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 $2^{\log n - i} \leq |\mathcal{B}_X(r) \cup \mathcal{B}_Y(r)| \leq 2^{\log n - i + 1}$. When $|\mathcal{B}_X(r) \cap \mathcal{B}_Y(r)|$ is $\frac{1}{2}$ of $|\mathcal{B}_X(r) \cup \mathcal{B}_Y(r)|$, for a 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 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 as follows: Suppose that r_k is bad. Then $|\mathcal{B}_X(dk) \cap \mathcal{B}_Y(dk)|$ is less than $\frac{1}{2}$ of $|\mathcal{B}_X(dk) \cup \mathcal{B}_Y(dk)|$. Notice that $\mathcal{B}_X(kd) \cap \mathcal{B}_Y(kd)$ contains $|\mathcal{B}_X((k-1)d) \cup \mathcal{B}_Y((k-1)d)|$, and since $|\mathcal{B}_X(kd) \cup \mathcal{B}_Y(kd)| \geq 2 |\mathcal{B}_X(kd) \cap \mathcal{B}_Y(kd)| \geq 2 |\mathcal{B}_X((k-1)d) \cup \mathcal{B}_Y((k-1)d)|$, we can say that $|\mathcal{B}_X(kd) \cup \mathcal{B}_Y(kd)| \geq 2 |\mathcal{B}_X((k-1)d) \cup \mathcal{B}_Y((k-1)d)|$. But this can happen at most $\log n$ times, since $|\mathcal{B}_X(r_1) \cap \mathcal{B}_Y(r_1)| \geq 2$ (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 $S_{0,0}$, it will clearly be pointed to by X and have a pointer to Y. □

To get the stretch bound, 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 $c * d(X, Y) \log^3 n$.

To provide load balancing, we let i range over all possible ID prefixes, and only search i's 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. CONCLUSION

We illustrate how to adapt to arriving and departing nodes in Tapestry, a location-independent overlay routing infrastructure with routing locality. This adaptation involves an efficient, distributed solution to the nearest-neighbor problem as well as a distributed algorithm for maintaining the prefix-based routing mesh. Both are presented for the first time in this paper. One of the salient properties of our system is that objects remain available, even as the network changes. Further, the cost of integrating new nodes is similar to that of 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.

8. REFERENCES

[1] AWERBUCH, B., AND PELEG, D. Concurrent online tracking of mobile users. In *Proc. of 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. *Israel J. Math 52* (1985), 46–52.

[5] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: Intl. 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. of the 29th Annual ACM Symp. on Theory of Comp.* (1997), pp. 609–617.

[7] COWEN, L. J. Compact routing with minimum stretch. In *Proc. of the 10th Annual ACM-SIAM Symp. on Discrete Algorithms* (January 1999), ACM.

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

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

[10] KARGER, D., AND RUHL, M. Find nearest neighbors in growth-restricted metrics. In *Proc. of the 34th Annual ACM Symp. on Theory of Comp.* (May 2002), pp. 741–750.

[11] KUBIATOWICZ, J., ET AL. OceanStore: An architecture for global-scale persistent storage. In *Proc. of ACM ASPLOS* (Nov 2000), ACM, pp. 190–201.

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

[13] PELEG, D., AND UPFAL, E. A tradeoff between size and efficiency for routing tables. In *Proc. of the Twentieth Annual ACM Symp. on Theory of Comp.* (May 1989), pp. 43–52.

[14] PLAXTON, C. G., RAJARAMAN, R., AND RICHA, A. W. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the 9th Annual Symp. on Parallel Algorithms and Architectures* (June 1997), ACM, pp. 311–320.

[15] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A scalable content-addressable network. In *Proc. of SIGCOMM* (Aug 2001), ACM, pp. 161–172.

[16] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware 2001* (November 2001).

[17] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of SOSP* (October 2001).

[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 *Proc. of SIGCOMM* (Aug 2001), ACM, pp. 149–160.

[19] THORUP, M., AND ZWICK, U. Approximate distance oracles. In *Proc. of the 33th Annual ACM Symp. on Theory of Comp.* (2001), pp. 183–192.

[20] THORUP, M., AND ZWICK, U. Compact routing schemes. In *Proc. of the 13th Annual Symp. on Parallel Algorithms and Architectures* (2001), pp. 1–10.

[21] ZHAO, B. Y., JOSEPH, A., AND KUBIATOWICZ, J. Locality-aware mechanisms for large-scale networks. In *Proc. of Workshop on Future Directions in Distributed Comp.* (June 2002).

[22] 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, UC Berkeley, Computer Science Division, April 2001.

RIGHTSLINK