Theory Comput. Systems OF1–OF36 (2004)
DOI: 10.1007/s00224-004-1146-6

Theory of Computing Systems

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,
Berkeley, CA 94720, USA
{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 to objects using names independent of the objects' physical locations. Two important properties of such 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 et al. [24].

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 [17], FarSite [3], CFS [12], and PAST [28]. To permit locality optimizations, it is important that the routing process use as few network hops as possible and that these hops be as short as possible.

* This research was supported by NSF Career Award #ANI-9985250, NFS ITR Award #CCR-0085899, an NSF Graduate Research Fellowship, UC MICRO Award #00-049, and NSF Grant #CCR-0105533.

[1] By locality we mean the ability to exploit local resources over remote ones whenever possible [36].

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 [22] and FreeNet [7].

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 [30], CAN [26], Pastry [27], and Viceroy [21], 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 a 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 polylogarithmic, the actual network latencies incurred by queries can be significantly more than those incurred by finding the object in the centralized directory.

An alternative solution is to broadcast an object's location to every node in the network. This allows clients to find the nearest copy of the object easily, 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 [35], [37]. Tapestry uses as a starting point the distributed data structure of Plaxton, Rajaraman, and Richa [24], which we refer to as the PRR scheme. Their proposal yields routing locality with balanced storage and computational load. However, it does not

[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.

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*

Several existing object location schemes exhibit routing locality, including those by Plaxton et al. [24], Awerbuch and Peleg [1], and Rajaraman et al. [25]. All of these provide the publication and deletion of objects with only a logarithmic number of messages and guarantee a low stretch, where stretch is defined as the ratio between the actual latency or distance to an object and the shortest distance. The PRR scheme finds objects with expected constant stretch for a specific class of network topologies while ensuring that no node has too many directory entries. Awerbuch and Peleg route within a polylogarithmic factor of optimal for general network topologies, but do not balance the load. Unfortunately, both the PRR and Awerbuch–Peleg schemes assume full knowledge of the participating nodes, or, equivalently, assume that the network is static. The RRVV scheme [25] balances the load, bounding the space at every node, and while only a polylogarithmic number of nodes need change when a node enters or leaves the network, it also does not give a method to find the nodes that need to be updated.

There is also an abundance of theoretical work on finding compact routing tables [2], [9], [23], [33] whose techniques are closely related to those in this paper. See [11] 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 ignore stretch. Chord [30] 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 [26] places objects into a virtual, high-dimensional space. Queries are routed along axes in this virtual space until they reach their destination. Pastry [27] 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. Viceroy [21] builds a constant-degree DHT based on the butterfly. A recent paper by Li and Plaxton [18] presents a simplified version of the PRR scheme that may perform well in practice. 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. Finally, all of these systems support the introduction and removal of nodes.

[3] A network topology gives a metric space.

Table 1. Comparison of Object Location Systems.*

Scheme	Insert cost	Space	Stretch, metric	Hops	Balanced?
CHORD [30]	$O(\log^2 n)$	$O(n \log n)$	–	$O(\log n)$	Yes
CAN [26]	$O(rn^{1/r})$	nr	–	$rn^{1/r}$	Yes
Pastry [27]	$O(\log^2 n)$	$O(n \log n)$	–	$O(\log n)$	Yes
Viceroy [21]	$O(\log n)$	$O(n)$	–	$O(\log n)$	Yes
This paper (Tapestry)	$\boldsymbol{O(\log^2 n)}$	$O(n \log n)$	–	$O(\log n)$	Yes
Awerbuch and Peleg [1]	–	$O(n \log^3 n)$	$O(\log^2 n)$, general	$O(\log^2 n)$	No
RRVV [25]	$O(\log^3 n)$	$O(n \log^3 n)$	$O(\log^3 n)$, general	$O(\log^2 n)$	Yes
PRR [24]	–	$O(n \log n)$	$O(1)$, 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	–	$O(n \log^2 n)$	$\boldsymbol{O(\log^3 n)}$, **general**	$O(\log^2 n)$	No

*In this table n is the number of nodes. For simplicity, we assume that the network diameter is polynomial in n, and that the number of objects is $O(n)$. Both stretch and hops refer to an object search. Space assumes that the object IDs occupy a constant number of bytes. Insert cost shows the number of hops or messages needed for node insertion; – means the system does not provide an algorithm. In RRVV the number of changes needed is polylogarithmic, but an algorithm to make the changes is not given. 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.

Recent peer-to-peer applications can locate objects in a dynamic network. Gnutella [22] utilizes a bounded broadcast mechanism to search neighbors for documents. FreeNet [7] 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].

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

Table 1 gives a summary of some of the previous results along with ours. Our contributions are in bold. Note that our result for general metrics can be improved using results of Thorup and Zwick [32] 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 is similar in spirit to that of Karger and Ruhl [15], who give an algorithm for answering nearest-neighbor queries in a similarly restricted metric space.[5]

The idea behind both the nearest-neighbor algorithm of Karger and Ruhl and the one presented here is to find the nearest neighbor by repeatedly finding some node halfway between the current node and the query node. If this is done $\log n$ times, one finds the closest node. The restricted metric spaces considered in both these papers mean that there is a substantial fraction of nodes at about the right distance, so halving the distance can be implemented by sampling from nodes within the correct radius. The difficulty is maintaining a structure to do the sampling in a dynamic network.

Karger and Ruhl suggested this general approach in [15] and then present a specific data structure to accomplish it. Their data structure uses a random permutation to maintain the random sampling—an approach is reminiscent of the Chord network infrastructure. Our search algorithm also aims to halve the distance at each step, but we build a different data structure with a different search algorithm. In particular, we use random names to build a tree (for load balancing purposes, many trees) on which we search. This set of trees is the same as the set of trees used in the object location system described in this paper, which means that our search algorithm can share the data structure with the object location algorithm.

Recently, Krauthgamer and Lee [16] developed a simple and deterministic nearest-neighbor data structure that has applications in a broader class of metric spaces. Their data structure is somewhat similar to the one described in this paper, but it is not yet clear if their data structure can be distributed in a load-balanced way.

We also prove that an alternate scheme by Plaxton et al. (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 [32] 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 gives algorithms for deletion. Section 6 discusses the issues in applying these theoretical results to physical networks. Section 7 gives a simple proof that the PRR v.0 scheme has polylogarithmic stretch for general metric spaces. Section 8 concludes.

[5] Clarkson also presented a very similar approach in [8].

2. The Tapestry Infrastructure

Tapestry [37], [35] is the wide-area location and routing infrastructure of OceanStore [17]. 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 refer to node identifiers as *node-IDs* and object identifiers as *globally unique identifiers* (GUIDs). For a string of digits α, let $|\alpha|$ be the number of digits in that string.

Tapestry inherits its basic structure from the data location scheme PRR [24]. 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 addressed with GUIDs and routed from node to node along neighbor links until an appropriate object pointer is discovered; the query is then forwarded along neighbor links to the destination node. Thus, every query ultimately resolves to a node-ID.

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, that is, 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, or $(|\beta| + 1)$. 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 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 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 the context is obvious, we 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

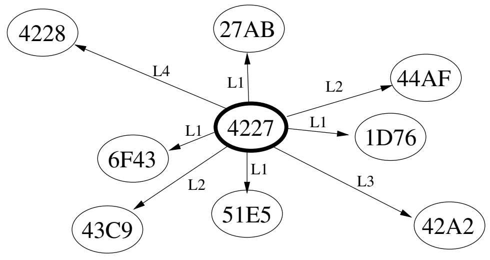

Fig. 1. Tapestry routing mesh. Each node is linked to other nodes via *neighbor links*, shown as solid arrows with labels. Labels denote which digit is resolved during link traversal. Here, node `4227` has an L1 link to `27AB`, resolving the first digit, an L2 link to `44AF`, resolving the second digit,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

of node-IDs. Membership in neighbor sets is limited by a constant parameter $R \geq 1$: $|\mathcal{N}^A_{\beta,j}| \leq R$, and of all the nodes that could be in the neighbor set, we choose the closest. Further, $|\mathcal{N}^A_{\beta,j}| < R$ implies $\mathcal{N}^A_{\beta,j}$ contains all (β, j) nodes. This gives us the following:

Property 1 (Consistency). If $\mathcal{N}^A_{\beta,j} = \emptyset$, 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 the destination node be B, with a node-ID equal to $\beta \equiv j_1 \circ j_2 \cdots j_n$. If ε is the empty string, then routing proceeds by choosing a succession of nodes: $A_1 \in \mathcal{N}^{A_0}_{\varepsilon,j_1}$ (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. This construction gives us locality, as described in the following property.

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 $\bigcup_{j \in [0,b-1]} \mathcal{N}^A_{\varepsilon,j}$, where ε represents the empty string. Section 3 discusses 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 $A \in \mathcal{R}_\psi$ 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 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 $A \in \mathcal{R}_\psi$. 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*, that is, pointers expire and objects must be republished (published again) at regular intervals. Republishing may be requested if something changes in the network.

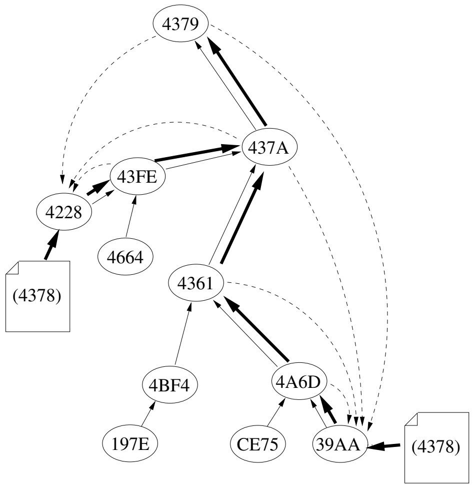

Fig. 2. Publication in Tapestry. To publish object `4378`, server `39AA` sends a publication request toward root, leaving a pointer at each hop. Server `4228` publishes its replica similarly. Since no `4378` node exists, object `4378` is rooted at node `4379`.

Queries for object ψ route toward *one* of the root nodes $A \in \mathcal{R}_\psi$ 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 is 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.

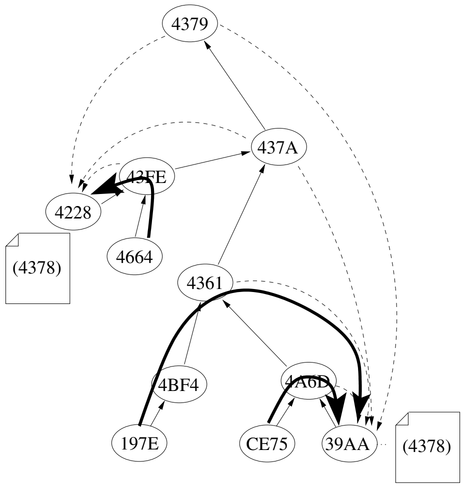

Fig. 3. Routing in Tapestry. Three different location requests. For instance, to locate GUID `4378`, query source `197E` routes toward the root, checking for a pointer at each step. At node `4361` it encounters a pointer to server `39AA`.

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 $A \in \mathcal{R}_\psi$ will (in the worst case) encounter a pointer for ψ upon reaching A. □

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 restricts its attention to metric spaces with a certain even-growth property: it assumes 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, Plaxton et al. [24] show the average distance traveled in locating an object is *proportional* to the distance from that object, that is, queries exhibit $O(1)$ stretch. Their data structure, however, ignores many issues important in practical systems. Tapestry is a simplification that is easier to implement and seems to provide low stretch in practice [35], [37].

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,B) produces an ordered list of the next hop toward the roots of ψ from node B.

In the PRR scheme, MapRoots(ψ) produces a single root node A 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. Roots reached in this way are considered *surrogate roots* of ψ.

Localized Routing Decisions. In a dynamic network, maintenance of routing pointers can be problematic. In Tapestry we have chosen to focus on Property 1 as our primary consistency constraint. Thus, in contrast to the original PRR scheme, we do not maintain extra route links to aid in locating root nodes. Instead, all routing decisions are made based on the current routing table, the source and destination GUIDs, and information collected along the route by the query (e.g., the number of digits resolved so far).

We highlight two variants on localized routing; others are certainly possible. Both of them proceed by routing one digit at a time toward the destination GUID, that is, each network hop resolves one additional digit toward the destination. Since there is no backtracking, these schemes are guaranteed to complete.

- **Tapestry Native Routing:** We route one digit at a time. 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 routed is a 3, and there is no entry, try 4, then 5, and so on.
- **Distributed PRR-Like Routing:** We route one digit at a time as follows:
  1. *Before first hole*: Route one digit at a time as above.
  2. *At first hole*: Route along an existing neighbor link that matches the desired digit in as many significant bits as possible. If there is more than one such route, pick the route with the numerically higher digit.
  3. *After first hole*: Always pick a routing-table entry with the numerically highest available digit.

  This technique routes to the root node with the numerically largest node-ID that matches the destination GUID in the most significant bits.

For both of these schemes, routing stops if the current node is the only node left at and above the current level in the routing table; the resulting node is the root node. Such localized schemes are simpler than PRR under inserts and deletes. In addition, the Tapestry Native Routing scheme 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, by Property 1. So both A' and B' send the message on a node with the same $(i+1)$st digit, a contradiction. $\square$

A similar proof is possible for the distributed PRR-like scheme. Localized routing may introduce additional hops over PRR; however, the number of additional hops is independent of n and in expectation is less than 2 [37]. 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 $\psi_0, \psi_1, \ldots, \psi_n$. Then, to route to root i, we surrogate route to ψ_i.*

2.4. *A Comparison between Tapestry and PRR*

In this section we quickly outline the differences between Tapestry and PRR. With the exception of some changes to the maintenance of object pointers, the key ways that Tapestry differs from PRR revolve around eliminating the requirements of a static network and adding provisions to handle faults in a graceful manner, both of which are required for a system deployable on real networks. To clarify the contributions of this work further, a few of the key differences are noted here.

First, Tapestry nodes keep pointers to all copies of a given object. In PRR a node stores at most one pointer to each object, regardless of how many copies are in the network. As a result, deleting an object is easier in Tapestry than in PRR, but comes with an increase in storage. Further, this property allows Tapestry applications to exploit multiple object replicas; queries can be multicast to every object of a given name or to a "close" subset of objects.

Second, PRR makes use of secondary neighbors in its core object location algorithms, while Tapestry uses them primarily for fault-resilience. When searching for an object, PRR searches on the primary and secondary neighbors before taking an additional hop toward the object root. It is worth noting that this is equivalent to publishing on all the secondary neighbors, but only searching the primary neighbors. In contrast, Tapestry keeps for every entry, a small number (in the current implementation, two) of additional backup neighbor links for fault-resilience. All publishing and search takes place only on the primary links.

Third, as described above, surrogate routing is different in the two systems. First, PRR assumes a static set of nodes with precomputed surrogates, while Tapestry's algorithms maintain the surrogate routes as part of the insertion process. PRR can be extended, however, with our dynamic algorithms while maintaining the same surrogate routing scheme. Tapestry surrogate routing does slightly better at load balancing of objects across the surrogate roots.

Finally, Tapestry is implemented and running, and downloads are available.

3. Building Neighbor Tables

Building the neighbor table is perhaps the most complex and interesting part of the insertion process, so we present it first. The problem is to build the neighbor sets, $\mathcal{N}^A_{\beta,j}$, for a new node A. These sets must satisfy Properties 1 and 2. This can be seen as solving the nearest-neighbor problem for many different prefixes. One solution is simply to use the method of Karger and Ruhl [15] many times, once for each prefix. This would essentially require each node to participate in $O(\log n)$ Karger–Ruhl data structures, one for each level of the neighbor table. This would require $O(\log^2 n)$ space.

The method we present below has lower network distance than a straightforward use of Karger and Ruhl (although the same number of network hops) and incurs no additional space over the PRR data structures.

As in [24], 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 let $|\mathcal{B}_A(r)|$ 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 that assumption is not needed for our extensions. Notice that our expansion property is almost exactly that used by Karger and Ruhl [15]. We also assume the triangle inequality in network distance, that is,

$$d(X, Y) \leq d(X, Z) + d(Z, Y)$$

for any set of nodes X, Y, and Z. Our bounds are in terms of network latency or network hops and ignore local computation in our calculations. None of the local computation is time-consuming, so this is a fair measure of complexity.

3.1. *The Algorithm*

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

Let a level-i node be a node that shares a length i prefix with α. Then, to go from the level-$(i + 1)$ list to the level-i list, we ask each node on the level-$(i + 1)$ list to give

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

  method GetNextList (neighborlist, level, NewNodeName, NewNodeIP)
1 nextList $\leftarrow \emptyset$
2 for $n \in$ neighborlist
3     temp $\leftarrow$ GetForwardAndBackPointers($n$, level)
4     AddToTableIfCloser [on $n$] (NewNodeName, NewNodeIP)
5     nextList $\leftarrow$ KeepClosestK(temp $\cup$ nextList)
6 return nextList
  end GetNextList
```

Fig. 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. The AcknowledgedMulticast function is described in Figure 8.

us all the level-i nodes they know of (we ask for both forward and backwards pointers). Note that each level-i node must have at least one level-$(i+1)$ node in its neighbor table, so following the backpointers of all level-$(i+1)$ nodes gives us all level-i nodes. We then contact these nodes, and sort them according to their distance from the inserting node. Each node contacted this way also checks to see if the new node should be added to its own table (line 4 of GetNextList). We then trim this list, keeping only the closest k nodes. If $b > c^2$, then Lemma 1 says there is some $k = O(\log n)$ such that with high probability the lists at each level contain exactly the k closest nodes.

We then use these lists to fill in the neighbor table. This happens in lines 4 and 7 of AcquireNeighborTable. More precisely, recall that level i of the table consists of nodes with the prefix $\alpha_{i-1} \circ j$, where α_{i-1} is the first $(i-1)$ digits of the node's prefix. To fill in level i of the neighbor table, we look in the level-$(i-1)$ list. For $j \in [0, b-1]$, we keep the closest $R\,(\alpha_{i-1}, j)$ nodes (R is defined in Section 2.1).[6]

3.2. *A Proof of Correctness*

Theorems 3 and 4 prove that with high probability, the above algorithm correctly creates the new node's neighbor table and correctly updates the neighbor tables of the existing nodes. Theorem 3 uses Lemmas 1 and 2 to show that the new node's table gets built correctly, and Theorem 4 argues that the tables of other nodes are updated correctly.

The following lemma shows that if GetNextList is given the k closest level-$(i+1)$ nodes, it finds the k closest level-i nodes.

Lemma 1. *If c is the expansion constant of the network, and $c^2 < b$, then given a list of the closest k level-$(i+1)$ nodes, we can find the k closest level-i nodes, for $k = O(\log n)$. In particular, if $k \geq (24(a+1)b \log n)/(1-c^2/b)^2$, the failure probability is bounded by $1/n^a$.*

Proof. 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 δ_{i+1} of the new node. If that is the case, then we will query A's parent, and so find A itself.

For the rest of the proof to work, we need that at least one of the k level-i nodes is also a level-$(i+1)$ node. The probability this is not true is $(1-1/b)^k \leq \exp(-k/b) \leq \exp(-(a+1)\log n) \leq 1/(n^{a+1})$. For the remainder of the proof, we assume at least one of the k level-i nodes is also a level-$(i+1)$ node. Then 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.

To complete the proof, we need $3\delta_i < \delta_{i+1}$ with high probability. This is the expected behavior; given a ball with k level-i nodes, doubling the radius twice gets no more than

[6] While the algorithm presented here is sensitive to failures, a slight modification can make the algorithm substantially more robust, see [13].

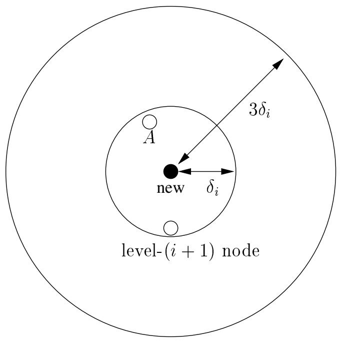

Fig. 5. Figure for Theorem 3. If $3\delta_i$ is less than δ_{i+1}, then A must point to a node within δ_{i+1}.

c^2k nodes, and so no more than $k(c^2/b)$ 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 argument into a proof.

First, recall that $c^2/b < 1$. Pick λ' and λ as follows:

$$\lambda = \tfrac{1}{2}(1 - c^2/b) < \tfrac{1}{2},$$
$$\lambda' = \lambda(2 - c^2/b) < 1.$$

Note that $\lambda' > \lambda$ and $(1-\lambda')b/c^2 = b/c^2(1-2\lambda) + \lambda = 1 + \lambda$. Notice that we can write k as a function of λ; in particular, $k = (6(a+1)b\log n)/\lambda^2$.

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.

Let L_{real} be the random variable representing the total volume of the ball (i.e., the number of nodes in the ball) containing k level-i nodes. If $3\delta_i < \delta_{i+1}$, then we are done, so in the rest of the proof we argue that the probability that $3\delta_i \geq \delta_{i+1}$ is small.

We use the fact that $\Pr[3\delta_i \geq \delta_{i+1}]$ is the same as

$$\Pr[3\delta_i \geq \delta_{i+1} \,|L_{\text{real}} > l\,] \cdot \Pr[L_{\text{real}} > l] + \Pr[3\delta_i \geq \delta_{i+1} \,|L_{\text{real}} \leq l\,] \cdot \Pr[L_{\text{real}} \leq l].$$

We will bound one term from each product.

We show $\Pr[L_{\text{real}} > l] \leq 1/n^{a+1}$

Let X_m be a random variable representing the number of level-i nodes in m nodes. Notice that the $\Pr[L_{\text{real}} > l]$ is bounded from above by $\Pr[X_l < k]$, since if $L_{\text{real}} > l$, then it must be that the closest l nodes to the new node do not contain k level-i nodes.

However,

$$\Pr[X_l < k] = \Pr[X_l < (1-\lambda')\mathrm{E}[X_l]].$$

Using a Chernoff bound, this is less than

$$\exp\left(\frac{-\lambda'^2 \mathrm{E}[X_l]}{2}\right) \le \exp\left(\frac{-\lambda'^2 k}{2}\right) \le \exp\left(\frac{-\lambda^2 k}{2}\right).$$

Substituting for k, this becomes

$$\exp\left(-\frac{6(a+1)b\log n\lambda^2}{2\lambda^2}\right) \le \exp(-3(a+1)\log n) \le \frac{1}{n^{a+1}}.$$

We show $\Pr[3\delta_i \ge \delta_{i+1} \,|\, L_{\text{real}} \le l\,] \le 2/n^{a+1}$

Consider the ball of radius $3\delta_i$ around the new node. If this ball contains k level-$(i+1)$ nodes (δ_{i+1} is smaller that $3\delta_i$), then 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 $c^2 l$ by (1) and the fact $L_{\text{real}} \le l$. Let Y_m be the number of $i+1$ nodes in m trials. Then, rewriting our goal with this notation, we wish to bound $\Pr[Y_{c^2 l} \ge k \,|\, X_l \ge k\,]$. (Notice that $Y_{c^2 l}$ is not independent of X_l.) We can write that

$$\Pr[A \,|\, B\,] = \frac{\Pr[A \cap B]}{\Pr[B]} \le \frac{\Pr[A]}{\Pr[B]},$$

so it suffices to bound $\Pr[Y_{c^2 l} \ge k]/\Pr[X_l \ge k]$. We already bounded the denominator, so next we wish to bound $\Pr[Y_{c^2 l} \ge k]$.

Since

$$\mathrm{E}[Y_{c^2 l}] = \frac{c^2}{b}\mathrm{E}[X_l] = \frac{kc^2}{b(1-\lambda')} = \frac{k}{1+\lambda}$$

we get that

$$\Pr[Y_{c^2 l} \ge k] = \Pr[Y_{c^2 l} \ge (1+\lambda)\mathrm{E}[Y_{c^2 l}]] \le \exp\left(\frac{-\lambda^2 \mathrm{E}[Y_{c^2 l}]}{3}\right).$$

However, $\mathrm{E}[Y_{c^2 l}] = k/(1+\lambda) \ge k/2$, so $\Pr[Y_{c^2 l} \ge k] \le \exp(-\lambda^2 k/6)$. Substituting k, we get that this is bounded by $\exp(-(a+1)\log n) = 1/n^{a+1}$.

So

$$\Pr[3\delta_i \ge \delta_{i+1} \,|\, L_{\text{real}} \le l\,] \le \frac{\Pr[Y_{c^2 l} \ge k]}{\Pr[X_l \ge k]} \le \frac{1/n^{a+1}}{1 - 1/n^{a+1}} \le \frac{2}{n^{a+1}}.$$

(As long as $n^{a+1} > 2$.)

Recall that we wish to bound $\Pr[3\delta_i \ge \delta_{i+1}]$, and we know that

$$\begin{aligned}
&\Pr[3\delta_i \ge \delta_{i+1}] \\
&\quad = \Pr[3\delta_i \ge \delta_{i+1} \,|\, L_{\text{real}} > l\,]\Pr[L_{\text{real}} > l] \\
&\qquad + \Pr[3\delta_i \ge \delta_{i+1} \,|\, L_{\text{real}} \le l\,]\Pr[L_{\text{real}} \le l]
\end{aligned}$$

$$\begin{aligned}&\le 1 \cdot 1/n^{a+1} + 2/n^{a+1} \cdot 1\\ &\le 1/n^a,\end{aligned}$$

where the last step follows as long as $n \ge 3$. □

Next, we show that given the k closest nodes matching in i digits, we can fill in level $(i+1)$ of the neighbor table if k is large enough. This means that given a list containing the k closest nodes with prefix β, for $k = O(\log n)$, there are at least R (β, j) nodes for each $j \in [0, b-1]$. (Recall that R was defined in Section 2.1.)

Lemma 2. *For $k = O(\log n)$ and $R = o(\log n)$ the list of the closest k β nodes to a given node A contains $\bigcup_j \mathcal{N}^A_{\beta,j}$ with high probability. In particular, for $k \ge 8(a+1)b\log n$, the probability it does not is less than $1/n^a$.*

Proof. For any given j, let X_j be a random variable representing the number of (β, j) nodes in the list. In expectation, this is $k/b = 8(a+1)\log n$. We want to bound $\Pr[X_j \le R]$, and since $R = o(\log n)$, we have $R \le \frac{1}{2}k/b$.

Thus,

$$\Pr[X_j \le R] \le \Pr[X_j \le \tfrac{1}{2}\mathrm{E}[X_j]] \le \exp\left(-\left(\frac{1}{2}\right)^2 \frac{\mathrm{E}[X_j]}{2}\right).$$

The last step uses a Chernoff bound. We can then simplify the last equation and say that $\Pr[X_j \le R] \le 1/n^{a+1}$.

Now, we apply a union bound over all the b possible j to get that the probability that any j has less than R nodes in the list is bounded by b/n^{a+1}, and since we can assume $b < n$, this gives us a bound of $1/n^a$, which is the desired result. □

Finally, we can combine these two lemmas to prove the following theorem:

Theorem 3. *If c is the expansion constant of the network, $b > c^2$ (where b is the digit size) and $R = o(\log n)$, then there is a $k = O(\log n)$ for which the algorithm of Figure* 4 *will produce the new node's correct neighbor table with probability $1/n^a$ for any constant a.*

Proof. First, by Lemma 1 there is a k_1 such that the probability that the ith list is incorrectly generated from the $(i+1)$st list is less than $1/(2n^{a+1})$. Since there are $\log n$ levels, the probability that any level fails is less than $(\log n)/(2n^{a+1}) \le 1/(2n^a)$.

Second, by Lemma 2, for some $k_2 = O(\log n)$, the probability that we are unable to fill the neighbor table with $R = o(\log n)$ neighbors from lists of length k_2 is less than $1/(2n^{a+1})$. Since there are $\log n$ levels to fill, the probability that any of the levels is left unfilled is bounded by $(\log n)/(2n^{a+1}) \le 1/(2n^a)$.

If we choose $k = \max(k_1, k_2)$, the probability that either of the lists are not correct or the table cannot be filled is bounded by $1/(2n^a) + 1/(2n^a) = 1/n^a$. This proves the theorem. □

The new node also causes changes to the neighbor tables of other nodes. We instruct any node that is a candidate for the new node's table to check if adding the new node could improve its own neighbor table. This happens in line 4 of GetNextList in Figure 4. It remains to show that with high probability, line 4 of GetNextList updates all nodes that need to be updated. In particular, we show 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 of level-i with high probability.

Theorem 4. *If a new node B is an (α, j) neighbor of A (so B is one of the R closest nodes to A with prefix $\alpha \circ j$), then with high probability A is among the $k = O(\log n)$ closest α nodes to B. In particular, for $k = 16abc \log n$, and $R = o(\log n)$, the probability A is not among the closest k nodes is $1/n^a$.*

Proof. We will show that the probability 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 in the neighbor table of A, there are less than R (α, j) nodes 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.

If A is not among the closest k nodes, then the ball around A of radius d contains no more than R (α, j) nodes, while the ball around B of radius d contains k nodes of prefix α. We will show the probability of this is very small. Since $R = o(\log n)$, for sufficiently large n, we can assume that $R \leq 2a \log n$.

Instead of arguing directly about the ball around B, we argue about the ball of radius $2d$ around A, since $\mathcal{B}_A(2d) \supset \mathcal{B}_B(d)$. More precisely, if we let $|\mathcal{B}_A(d)|_\alpha$ denote the number of α nodes in the ball of radius d around A, then we want to argue that the

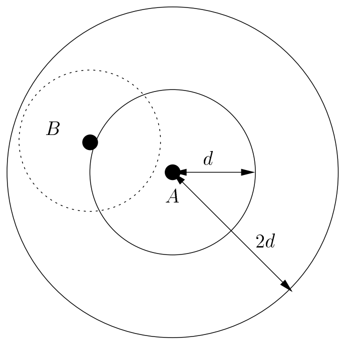

Fig. 6. Figure for Theorem 4. The larger ball around A contains $O(\log n)$ nodes, while the smaller ball contains none.

probability that $|\mathcal{B}_A(d)|_{(\alpha,j)} \le R$ and $|\mathcal{B}_A(2d)|_\alpha \ge k$ is small. To do this, we have two cases, depending on $|\mathcal{B}_A(d)|$. Let l_{real} be the number of nodes in the smaller ball around A (or $|\mathcal{B}_A(d)|$) and let $l = 8(a/p)\log n$ where p is the probability a node is an (α, j) node. (Note that l_{real} is not a random variable.) Finally, let $k = 16abc\log n$. (Recall that b is the base of the logarithm and c is the expansion constant of the network.)

Case 1: $l_{\text{real}} > l$. Intuitively, this case is unlikely because if the smaller ball around A has many nodes, we expect there to be many (α, j) nodes.

More formally, let X_m be the number of (α, j) nodes found in m trials. Then $X_{l_{\text{real}}}$ is the random variable representing the number of (α, j) nodes found in the closest l_{real} nodes to A. In this case we want to bound the probability that $X_{l_{\text{real}}} \le R$. Then we can say that $\Pr[X_{l_{\text{real}}} \le R] \le \Pr[X_l \le R]$. Further, $\mathrm{E}[X_l] = p(8a/p)\log n = 8a\log n$, so $\Pr[X_l \le R] \le \Pr[X_l \le (1-\frac{1}{2})\mathrm{E}[X_l]]$. Using a Chernoff bound, this is less than $\exp(-\frac{1}{2}^2\mathrm{E}[X_l]/2)$, and substituting, this is less than $\exp(-\frac{1}{2}^2 8a\log n/2) \le \exp(-a\log n) = 1/n^a$.

Case 2: $l_{\text{real}} \le l$. In this case we argue that the ball of radius $2d$ around A contains more than k nodes with probability less than $1/n^a$. However, since the ball around B of radius d is contained in this ball, this will imply that the ball around B of radius d contains more than k nodes with probability less than $1/n^a$.

Let Y_m be a random variable representing the number of α nodes in m trials. We wish to bound

$$\Pr[|\mathcal{B}_A(2d)|_\alpha \ge k],$$

but this is less than or equal to $\Pr[Y_{cl} \ge k]$, since $\mathcal{B}_A(2d)$ contains at most cl nodes. Then

$$\Pr[|\mathcal{B}_A(2d)|_\alpha \ge k] \le \Pr[Y_{cl_{\text{real}}} \ge k] \le \Pr[Y_{cl} \ge k].$$

Recalling that $\mathrm{E}[Y_{cl}] = (pb)c(8a/p)\log n$, $k = 2\mathrm{E}[Y_{cl}]$, so again using a Chernoff bound, we can write

$$\Pr[Y_{cl} \ge 2\mathrm{E}[Y_{cl}]] \le \exp(-\tfrac{8}{3}abc\log n) \le \exp(-a\log n).$$

Since the probability of each case is bounded by $1/n^a$, the overall probability is also bounded by $1/n^a$, completing the proof. □

To make sure the probability of failing to get either the new node's table or correctly update the tables of established nodes is less than $1/n^a$ for some a, we combine the results in the following way. Let k_1 be large enough that the probability a mistake is made in building the neighbor table is less than $1/(2n^a)$ (this is possible by Theorem 3), and choose k_2 large enough that the probability that the algorithm misses an update to another node's table is less than $1/(2n^a)$ (possible by Theorem 4). Finally, choose $k = \max(k_1, k_2)$, and the probability that the algorithm of Figure 4 fails to perform the correct updates will be less than $1/(2n^a) + 1/(2n^a) \le 1/n^a$.

3.3. *Running Time*

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 local computation.)

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. However, this analysis can be tightened. Using the techniques of Theorems 3 and 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.

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 in addition to updating the neighbor tables correctly, 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 show that if Properties 1, 2, and 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 shows 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 4 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.

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

Fig. 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.

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 is 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 apply. 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

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

Fig. 8. Acknowledged Multicast. It runs FUNCTION on all nodes with prefix α.

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 α. 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 are 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 the state at all the participating nodes, but this is beyond the scope of this paper.

4.2. *Redistributing Object Pointers*

Recall that objects 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 toward 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

```
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
```

Fig. 9. OptimizeObjectPtrs and its helper function.

path skips node B, then node B will still be left with a pointer to the object. Further, the simple republish may do extra work updating pointers that have not changed.

Instead, a nodes with a new forward route 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. Figure 9 shows the two methods that are needed to implement this procedure.

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 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 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. However, we want to do this in such a way that the surrogate does not need to keep the 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

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

Fig. 10. Misrouting and route correction. Misrouting and route correction are used to keep objects available even during insertion.

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.

Finally, it is possible for a request for a non-existent object to loop until the insertion is complete. We address this problem by including information in the message header about where the request has been, allowing the system to detect and prevent loops. Since the number of hops is small, this is not an unreasonable overhead.

4.4. *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 re-inserting the node, but before the re-insertion occurs, objects may be unavailable. This is a serious problem, and this section presents our solution. (Recent work by Liu and Lam [20] also addresses this problem; their solution has the advantage that it does not requires that non-joining nodes maintain the state about on-going joins.)

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, that is, it has no holes in its neighbor table that can be filled by other core nodes in the network, and it cannot fill holes in the neighbor tables of core 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 to 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 finishes remain core nodes. We also add the requirement that any multicast, including those used in insertion, must start at a core node.

Two operations are *simultaneous* if there is a point in time when both operations are ongoing. This straightforward 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. In the following the *hole* that a new node fills is the slot in the surrogate's neighbor table for which there was no available core node to perform the multicast operation.

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.*

This is a proof by induction. We order the nodes by when they finish their multicasts. By the induction hypothesis, all nodes that have finished before A satisfy the theorem, and we prove the same is true of A. (Note that we are not assuming that there are no ongoing multicasts when A starts its multicast.) We start by proving a series of lemmas. Our first lemma is simple, but important. Lemma 3 states that simultaneously inserting nodes cannot interfere with one another's access to core nodes.

Lemma 3. *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 toward X but did not. Further, suppose that the prefix B received with the multicast was α. If B did not send the multicast toward 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 an (α, j) node in its table. However, 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 share prefix α. Further, since we know that X was in the network before B began its multicast, B's multicast consists of nodes with prefix α. However, 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, since X is an (α, j) node, B must have such a node in its table. Contradiction.

Although Theorem 6 uses Lemma 3, Case 2 is not circular: node B was inserted *before* the point in time that we use it here. □

It remains to deal with the case where two insertions conflict. We first introduce the notion of a pinned pointer. An (α, j) pointer to node A stored at node X is *pinned* 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 pinned pointer and sends the multicast to one unpinned pointer and all pinned pointers. When X receives acknowledgments from all recipients, X unlocks the pointer. Finally, X must keep at least one unpinned pointer and all pinned pointers. If this is done, then X will reach all (α, j) nodes it knows about without having to store them all. Intuitively, the unpinned pointer can reach all other unpinned pointers so unpinned pointers are all equivalent, while the pinned pointers are not well-enough connected to be reachable via multicast.

Lemma 4. *A multicast through an unpinned (α, j) pointer at node X reaches all other nodes that have or had unpinned (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 unpinned pointer and all the pinned pointers.

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

Lemma 5. *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 to A, since they fill the same hole. Finally, since X does not send an acknowledgment until A returns an acknowledgment of B's multicast, A has been informed by the time B's multicast finishes. We then apply this same argument with the roles of A and B reversed. □

We are not yet done. 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.

```
   method AcknowledgedMulticast($\alpha$, Function, holebeingfilled, watchlist,
   NewNodeIP)
1  watchlist $\leftarrow$ CheckForNodesAndSend(watchlist, NewNodeIP)
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 [on GetIP(neighbor)] ($\alpha \circ i$, Function,
             holebeingfilled, watchlist, NewNodeIP)
7  else
8    apply Function
9    $\mathcal{S} \leftarrow$ MulticastToFilledHole(holebeingfilled, Function, watchlist,
     NewNodeIP)
10 wait $\mathcal{S}$
11 SendAcknowledgement()
   end AcknowledgedMulticast
```

Fig. 11. Acknowledged multicast with the watch list. This version of acknowledged multicast handles simultaneous insertions.

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 16 bits. This new version is shown in Figure 11. Using this new multicast, we get Lemma 6.

Lemma 6. *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 an (α, j) hole for $i \neq j$) , then they multicast to the same prefix α. By Lemma 3 we know these nodes are reached, and we can apply Lemma 6, once with A in the theorem being A of the lemma, and once with A in the theorem as B in the lemma.

Case 2. If A and B fill different holes on different levels, then there are core α nodes in the network, and by Lemma 3 we know these nodes are reached. Given that, we again apply Lemma 6.

Case 3. If A and B fill the same hole on the same level, then there might not be a core node with prefix α so the preceding arguments fail. In this case we rely on Lemma 5, which says that if the two multicasts are not serialized, each will find out about the other before their multicasts are complete.

This completes the proof. $\square$

Discussion. Note that this parallel insertion algorithm is *lock-free*; although the multicast must start from a core node, a core node can perform multicasts for many inserting nodes. The process of pinning pointers does not impede forward progress of insertion.

However, a side effect of this lock-free behavior is that a new node may receive multicasts from several other inserting nodes. Fortunately, this effect 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.

4.5. *Running Time Analysis*

The total number of hops is $O(\log^2 n)$ with high probability.[7] Finding the surrogate is no more costly than searching for an object pointer, and Plaxton et al. [24] argue that finding an object pointer requires $O(d)$ network traffic (and $O(\log n)$ hops). Multicast takes time $O(kd)$ where k is the number of nodes reached. However, k will be small in expectation, and bounded by $\log n$ with high probability. 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 objects is at most $O(md)$. This gives a total traffic of $O(md \log n)$ for object pointer relocation.

5. Node Deletion

In this section we present algorithms that help maintain our invariants 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 the preferred mode of deletion that permits the infrastructure to maintain availability of objects by fixing neighbor links and object pointers. An *involuntary* delete occurs when a node ceases to

[7] Determining k dynamically, this cost can be reduced to $O(\log n)$ with high probability [14].

participate in the network without warning, due to a node failure, a network failure, or an attack. Note that it is unreasonable to hope that all deletes are voluntary deletes, and we present an algorithm for this case only for completeness. In real networks, nodes and links will typically fail without warning, so involuntary delete is the common case. We discuss this case in Section 5.2.

5.1. *Voluntary Delete*

When node A decides to leave the network, it ideally removes itself in a way that gives the infrastructure time to adapt its routing mesh and object pointers to maintain object availability. A begins by sending its intention to leave the network to all nodes on its backpointers list (all nodes which currently point to A somewhere in their routing table). Along with this notification, A sends along a potential replacement for itself on each routing level. On each such node (N), links to A are marked as "leaving."

Removing the link to A could leave N with an incorrect hole in its routing table (breaking Property 1). This problem is mitigated by any existing secondary pointers backing up A and by potential replacements A sends with its notification. Node N may still wish to run the nearest-neighbor algorithm to tune the neighbor table.

When this initial notification is received by node N, it republishes any local object pointers which normally route through A as if A did not exist. Any incoming queries still route normally to A while it is marked as "leaving." Publish operations, however, route to both A and its replacement. See Figure 12.

After node A sends out its initial notification messages, it examines local object pointers for which it is the root, and forwards them to their respective surrogate nodes. Once all of these objects have found new root nodes and acknowledgments are received by A, objects that were rooted at A are now reachable through new surrogates. Thus availability is guaranteed. Node A then sends out a final delete notification to its backpointers, telling them to delete A from their routing tables completely. After all such nodes have responded, A disconnects. If it is allowable for objects to be temporarily unavailable, much of this work can be skipped, and the notification can happen in one phase rather than two.

5.2. *Involuntary Delete*

Involuntary deletion occurs when any failure prevents a node from performing normal Tapestry operations. For simplicity, we consider here only complete failures such as network partitions, hardware failures, or complete system halts. In these scenarios we

```
  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
```

Fig. 12. Voluntary Delete. This shows what a node should do when it leaves the network.

would like the rest of the Tapestry network to detect this node's failure and recover as much as possible to maintain object availability and full reachability of the routing mesh.

We propose that unexpected deletes be handled lazily. That is, when a node N 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. In the process of fixing its state, however, N may hint to other nodes that their state may be out of date. Deletion can be detected by soft-state beacons [37] or when a node sends a message to a defunct node and does not get a response.

When node N detects a faulty node, it should first remove the node from its neighbor table and find a suitable replacement. If this produces a hole in the table, N will have to find a replacement, to ensure Property 1 is maintained. Otherwise, N has several options depending on how good the replacement must be. It 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. 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 its routing table, we must either find a replacement, or determine that none exists. To do so, we could use a multicast to all nodes sharing the same prefix of N and the dead node. While this is a workable solution, the multicast algorithm assumes all tables are complete, and may not reach a given node if some table along the path is incomplete. We can do slightly better. Liu and Lam [20] present a notification algorithm similar to multicast that solves this problem. Their algorithm has the property that if the node is in the table of *any* contacted node, then it will be returned to the node starting the multicast. Furthermore, their notification algorithm also requires only the starting node to maintain the state. This makes it ideal for this application.

One concern is that if a node N disappears, every node that used to point to N might start such a search, causing more traffic than is desirable. At the cost of centralization, we can pick one node (say the surrogate of the departed node) to perform the search, and have the surrogate return the answer when done.

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. Realistic Deployment

In previous sections we presented algorithmic solutions for maintaining Properties 1 and 2 during membership changes to the overlay network. These algorithms assume that the network satisfies the expansion property and that distances are unchanging. In this section we reconsider these assumptions and other issues in the context of physical networks such as the Internet.

6.1. *The Power of Indirection*

Tapestry does not replicate data, only references to data. In systems that do not consider stretch (and consider instead, for example, network hops), one can always add a level of indirection and view data items as pointers at a cost of only one additional hop; consequently, this distinction may not seem significant. However, the maintenance of pointers within the network (rather than at application level) is a powerful abstraction. Further, adding a level of indirection can drastically change the stretch properties even when it does not significantly change the number of network hops.

In addition, an object location system that allows arbitrary object placement is extremely flexible. Tapestry enables applications to choose their own data placement policies, such as placing replicated objects near hotspots or on reliable nodes. Further, it automatically makes use of close resources when possible. This reduces network traffic and bandwidth, and may improve reliability as well, since less links are traversed on an access to a nearby object.

6.2. *Physical Network Topologies*

One important assumption for this paper is the expansion property of (1). Most real networks cannot be described this simply. Several research projects have tried to create accurate models of the current Internet. One of the more widely accepted models is the transit-stub network model [34]. Transit-stub networks may or may not have the expansion property, depending on the layout of nodes inside the stub. Ideally, we want to provide good performance on such networks regardless of the intra-stub layout.

The algorithm of Section 3 may not find the nearest neighbor if the expansion constant of the network is too large. However, a modification of the algorithm which chooses k dynamically can guarantee to give the right answer always, though it may take a long time when the expansion is high (see [14]). An algorithm without a strong dependence on the expansion factor seems unlikely, since high expansion implies high dimension, and high dimension neighbor searches are known to be hard.

Castro et al. [6] explore the question of the expansion property by running simulations on several different topology models. Their results suggest that even when the expansion property does not hold, it can be useful in guiding the design of overlay networks. In the context of this paper, our nearest-neighbor algorithm seems to continue to perform well with real network topologies [35].

6.3. *Locality Enhancement*

When the expansion property does not hold, the routing stretch may become quite high. Note, however, that the system will always find an object after $O(\log n)$ hops, and so in the worst case it still competes with systems which are not locality-aware. Nonetheless, we can expand the pointer placement mechanisms described in Section 2 to enhance the locality and reduce stretch.

For instance, because latency differences between intra-stub paths and inter-stub paths can be an order of magnitude or greater, we would like to ensure that an object locate request never leaves the originating stub if there is a copy of the object somewhere inside the stub. We propose an optimization for object publication and locate operations that makes an effort to limit the operation to the local stub domain.

Assume, for the moment, that Tapestry nodes can detect whether the next hop is within the same stub network. Then we can ensure that a message never leaves the stub when the desired object is inside the stub. When object publication is about to route out of the local network, it spawns off a “local branch” publish message which treats the local network as its entire domain. While the original routes out to the wide-area, local publish uses surrogate routing to route to a local root, where it terminates. If an object exists in the local network, then the request will terminate at or before the local object root. Otherwise, it examines the current node’s ID to determine when surrogate routing started, and resumes at that hop to continue with normal object location.

For example, suppose node 1224 is trying to find an object labeled 1234. If a 123X node exists, but lies outside the local network, node 1224 uses surrogate routing inside the local stub, and tries to send the message to the closest 124X node in the stub. If none exists, it sends to the closest 125X node in the stub, and so on, until it finds a definitive local root node for 1234. If the object is not found before or at the local stub root, then node 1224 resumes normal routing outside the stub (to 123X). On the publish side, if a local node is publishing the object with ID 1234 and discovers the next hop is outside of the local network, it forwards two publish messages for the same ID. One continues as a normal publish outside of the network, while the other is restricted to the local-area, and publishes with surrogate routing to a local object root. In practice, it may not be possible to determine exactly whether a node is in the same stub or not. However, this can probably be guessed by setting a local latency threshold and marking nodes further than the threshold as outside the stub. Details and simulation for this and other locality-enhancement schemes appear in [31].

The net effect of this optimization is that queries for objects within a given stub network are always resolved without routing outside the stub network. The tradeoff is that queries for remote objects will pay the price of additional surrogate routing hops inside the stub network; however, these are local hops and surrogate routing lasts less than two hops in expectation [37]. Consequently, systems should see significant performance gains as a result of eliminating wide-area network traffic.

6.4. *Continual Optimization*

In the real Internet, routes frequently change due to a variety of factors, some of which are configuration changes to Border Gateway Protocol (BGP), changes in packet forwarding policy between Internet Service Providers (ISPs), or recalculation of IP routes after router failures inside Autonomous Systems (ASs). Therefore, network distance can change over time, potentially thwarting our efforts to provide locally optimal routes at each hop. In this section we discuss some heuristics to adapt our network mesh to the underlying network distance instability better. These optimizations trade computation and network traffic for more up-to-date network structure.

The simplest thing that we can do is to adjust routing table entries. Recall that for any entry, there are R neighbor links; we periodically adjust which of these neighbors is the primary. At the opposite extreme, we can invoke periodic repetitions of the complete nearest-neighbor algorithm.

A third option is for a node to record the identities of all the nodes contacted in its search for a nearest neighbor—at most $O(\log^2 n)$ nodes. Then we can optimize one level or one entry at a time. To optimize level i, for example, we contact all the level-$(i+1)$ nodes that were contacted during the original building process, and rebuild level-i. Since this should be an infrequent operation, the additional storage can be flushed to disk to minimize memory overhead. The frequency can be set for a given type of network, or can be dynamically triggered when nodes discover significant changes in their expected node to node performance.

A fourth option is to use local sharing of information. That is, periodically a node sends to its level-i neighbors a copy of its level-i neighbor table. The receiving nodes repeat distance measurements to compare those neighbors with its own, replacing further away nodes where appropriate. This is the same idea as the heuristic neighbor table building algorithms in [27] and [37].

In all these cases, when a new primary neighbor has been chosen, the node needs to move some object pointers. This can be done efficiently as described in Section 4.2. Note that such pointer movement can often be deferred until a later time, since it does not affect the correctness of the object location process.

6.5. *Reliable Messaging and Sybil Attacks*

Finally, we mention two more significant factors to consider in real deployment of systems such as Tapestry. First, we reconsider our assumption that all messages arrive reliably at their destinations in order. In the Internet, packets can be dropped, connections can be broken, and communication reliability cannot be guaranteed. This is exacerbated by the large scale nature of systems like Tapestry.

To provide resilience against unpredictable failures, we use soft-state—state that is not needed for correctness. For example, nodes use periodic heartbeat messages to detect node and link failures, while objects are republished at regular intervals in order to maintain high availability. Soft-state mechanisms limit the loss of availability due to failures and most attacks, while simplifying our algorithms. The efficacy of these techniques is illustrated in [35]. To achieve greater resiliency to failure, we can invoke epidemic propagation mechanisms to synchronize groups of pointers among alternate root sets; such techniques are the topic of ongoing research.

Second, we need to be aware of malicious users and potential attacks on the system (Sit and Morris [29] describe some of these attacks). One particularly devastating attack occurs when a single attacker impersonates a large number of physical nodes in an attempt to gain control over key positions in the routing mesh. By generating enough randomized node IDs, an attacker would eventually obtain IDs that position it around an intended target node in the mesh. Attacks can then be carried out to deny service or manipulate and spoof communication.

These attacks, referred to as Sybil Attacks [10], are extremely hard to detect and circumvent. Given enough physical resources, an attacker can be hard to overcome. One viable solution is to use a centralized certificate authority to allocate node IDs. Its centralized nature, however, might lead to problems in scalability. This problem, like several others related to security, remains unsolved and is the subject of active research. (See [5] for some approaches to this problem.)

7. Object Location in General Metric Spaces

In this section we take a slightly different tack and allow an arbitrary metric space S, but do not make the scheme dynamic. We show how to route to an object with polylogarithmic stretch and $O(|\text{ID}| \log^2 n)$ average space, where $|\text{ID}|$ is the size of an object ID. We remark that this is the strawman scheme proposed by Plaxton et al. [24] without load balancing, and is similar to the scheme of Thorup and Zwick [32]. The proof is reminiscent of the metric embedding results of Bourgain [4] and Linial et al. [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. The following theorem is the key to showing stretch bounds.

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 $d(S_{i*,j}, X) \leq d(X, Y) \log n$ with high probability. Moreover, the average space used by the data structure is $O(\log^2 n)$.*

Proof. Let $\mathcal{B}_X(r)$ be the ball around X of radius r, that is, all the nodes within distance r of X. Now, consider a sequence of radii such that $r_k = kd$ for $k \in [1, \log n]$ and $d = d(X, Y)$. 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 half of $|\mathcal{B}_X(r) \cup \mathcal{B}_Y(r)|$, for a given j, with constant probability there will be 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. 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 now argue that some r_k is good. Suppose that r_k is bad. Then $|\mathcal{B}_X(dk) \cap \mathcal{B}_Y(dk)|$ is less than half 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

$$\begin{aligned} |\mathcal{B}_X(kd) \cup \mathcal{B}_Y(kd)| &\geq 2|\mathcal{B}_X(kd) \cap \mathcal{B}_Y(kd)| \\ &\geq 2|\mathcal{B}_X((k-1)d) \cup \mathcal{B}_Y((k-1)d)|, \end{aligned}$$

we can say that

$$|\mathcal{B}_X(kd) \cup \mathcal{B}_Y(kd)| \geq 2|\mathcal{B}_X((k-1)d) \cup \mathcal{B}_Y((k-1)d)|.$$

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

Note that we have assumed implicitly that the distance to the nearest $S_{i+1,j}$ is always less than the distance to $S_{i,j}$. This may not be strictly true. To make it true, we can require that $S_{i,j} \subset S_{i+1,j}$. Doing this would change the probability of a point being in $S_{i,j}$ only slightly, so the result still holds.

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 maintain this data structure efficiently.

8. Conclusion

We illustrate how to adapt to arriving and departing nodes in Tapestry, a location-independent overlay routing infrastructure with routing locality. We describe an efficient, distributed solution to the nearest-neighbor problem as well as a distributed algorithm for maintaining the prefix-based routing mesh. 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.

Acknowledgments

We thank Sean Rhea for his careful reading of this paper. We also thank Kevin Chen, Megan Thomas, and the members of the OceanStore group and the anonymous reviewers for their insightful comments.

References

[1] Awerbuch, B., and Peleg, D. Concurrent online tracking of mobile users. In *Proc. SIGCOMM* (Sept. 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. SIGMETRICS* (June 2000), pp. 34–43.

[4] Bourgain, J. On Lipschitz embedding of finite metric spaces in Hilbert space. *Israel Journal of Mathematics*, **52** (1985), 46–52.

[5] Castro, M., Druschel, P., Ganesh, A., and Rowstron, A. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the 5th Symposium on Operating Systems Desgin and Implementation* (Dec. 2002), pp. 299–314.

[6] Castro, M., Druschel, P., Hu, Y. C., and Rowstron, A. Exploiting network proximity in peer-to-peer overlay networks. In *Proceedings of the International Workshop on Future Directions in Distributed Computing* (June 2002), pp. 52–55.

[7] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. Freenet: a distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*: *International Workshop on Design Issues in Anonymity and Unobservability* (H. Federrath, ed.). Lecture Notes in Computer Science 2009, pp. 46–66, Springer-Verlag, Berlin, 2001.

[8] Clarkson, K. L. Nearest neighbor queries in metric spaces. In *Proceedings of the* 29*th Annual ACM Symposium on Theory of Computing* (May 1997), pp. 609–617.

[9] Cowen, L. J. Compact routing with minimum stretch. In *Proceedings of the* 10*th Annual ACM–SIAM Symposium on Discrete Algorithms* (Jan. 1999), pp. 255–260.

[10] Douceur, J. The Sybil attack. In *Proc. IPTPS* (Mar. 2002), pp. 251–260.

[11] Gavoille, C. Routing in distributed networks: overview and open problems. *ACM SIGACT News*, **32**(1) (Mar. 2001), 36–52.

[12] Gopal, B., and Manber, U. Integrating content-based access mechanisms with hierarchical file systems. In *Proc. ACM OSDI* (Feb. 1999), pp. 265–278.

[13] Hildrum, K., and Kubiatowicz, J. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *Proceedings of the* 17*th International Symposium on Distributed Computing* (Oct. 2003), pp. 321–336.

[14] Hildrum, K., Kubiatowicz, J. D., and Rao, S. Another way to find the nearest neighbor in growth-restricted metrics. Tech. Rep. UCB/CSD-03-1267, Computer Science Division, UC Berkeley, Aug. 2003.

[15] Karger, D., and Ruhl, M. Find nearest neighbors in growth-restricted metrics. In *Proceedings of the* 34*th Annual ACM Symposium on Theory of Computing* (May 2002), pp. 741–750.

[16] Krauthgamer, R., and Lee, J. Navigating nets: simple algorithms for proximity search. In *Proceedings of the* 15*th Annual ACM–SIAM Symposium on Discrete Algorithms* (Jan. 2004), pp. 791–801.

[17] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., 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 *Proc. ACM ASPLOS* (Nov. 2000), pp. 190–201.

[18] Li, X., and Plaxton, C. G. On name resolution in peer-to-peer networks. In *Proceedings of the Second ACM International Workshop on Principles of Mobile Computing* (Oct. 2002), pp. 82–89.

[19] Linial, N., London, E., and Rabinovich, Y. The geometry of graphs and some of its algorithmic applications. In *Proceedings of the IEEE Symposium on Foundations of Computer Science* (1994), pp. 577–591.

[20] Liu, H., and Lam, S. S. Neighbor table construction and update in a dynamic peer-to-peer network. In *Proc. IEEE ICDCS* (May 2003), pp. 509–519.

[21] Malkhi, D., Naor, M., and Ratajczak, D. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing* pp. 183–192. ACM Press, New York, 2002.

[22] Oram, A., Ed. *Peer-to-Peer, Harnessing the Power of Disruptive Technologies*. O'Reilly, Cambridge, MA, 2001.

[23] Peleg, D., and Upfal, E. A tradeoff between size and efficiency for routing tables. In *Proceedings of the* 21*st Annual ACM Symposium on Theory of Computing* (May 1989), pp. 43–52.

[24] Plaxton, C. G., Rajaraman, R., and Richa, A. W. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the* 9*th Annual Symposium on Parallel Algorithms and Architectures* (June 1997), pp. 311–320.

[25] Rajaraman, R., Richa, A. W., Vöcking, B., and Vuppuluri, G. A data tracking scheme for general networks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures* (July 2001), pp. 247–254.

[26] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. A scalable content-addressable network. In *Proc. SIGCOMM* (Aug. 2001), pp. 161–172.

[27] Rowstron, A., and Druschel, P. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms* (Nov. 2001), pp. 329–350.

[28] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP* (Oct. 2001), pp. 188–201.

[29] Sit, E., and Morris, R. Security considerations for peer-to-peer distributed hash tables. In *Proceedings of the First International Workshop on Peer-to-Peer Systems* (Mar. 2002), pp. 261–269.

[30] 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. SIGCOMM* (Aug. 2001), pp. 149–160.

[31] Stribling, J., Hildrum, K., and Kubiatowicz, J. D. Optimizations for locality-aware structured peer-to-peer overlays. Tech. Rep. UCB/CSD-03-1266, Computer Science Division, UC Berkeley, Aug. 2003.

[32] Thorup, M., and Zwick, U. Approximate distance oracles. In *Proceedings of the* 33*rd Annual ACM Symposium on Theory of Computing* (July 2001), pp. 183–192.

[33] Thorup, M., and Zwick, U. Compact routing schemes. In *Proceedings of the* 13*th Annual Symposium on Parallel Algorithms and Architectures* (July 2001), pp. 1–10.

[34] Zegura, E. W., Calvert, K., and Bhattacharjee, S. How to model an internetwork. In *Proc. IEEE INFOCOM* (Mar. 1996), vol. 2, pp. 594 –602.

[35] Zhao, B. Y., Huang, L., Rhea, S. C., Stribling, J., Joseph, A. D., and Kubiatowicz, J. D. Tapestry: a global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, **22**(1) (2004), 41–53. Special Issue on Service Overlay Networks.

[36] Zhao, B. Y., Joseph, A., and Kubiatowicz, J. Locality-aware mechanisms for large-scale networks. In *Proceedings of the Workshop on Future Directions in Distributed Computing* (June 2002), pp. 80–83.

[37] 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, Computer Science Division, UC Berkeley, Apr. 2001.

Received October 30, 2002, *and in revised form May* 5, 2003, *and in final form August* 19, 2003.
Online publication March 15, 2004.