Finding Nearby Objects in Peer-to-Peer Networks

by

Kirsten Weale Hildrum

B.S. (University of Washington) 1998
B.S. (University of Washington) 1998

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor John Kubiatowicz, co-Chair
Professor Satish Rao, co-Chair
Professor Dorit Hochbaum

Spring 2004

Abstract

Finding Nearby Objects in Peer-to-Peer Networks

by

Kirsten Weale Hildrum

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor John Kubiatowicz, co-Chair

Professor Satish Rao, co-Chair

A peer-to-peer object location system is an evolving set of computers cooperating to store objects. A reasonable system should easily adapt when computers join or leave the network (self-organization), reliably find objects (completeness), and ensure that no computer works too hard (load balance). Searches in this network should find nearby copies of objects when possible: a searcher in Berkeley looking for an object on the Berkeley subnetwork should find the object without ever sending a message outside of Berkeley.

In this thesis, we describe the first techniques to maintain these properties. Our performance depends on an upper bound on the growth rate of the network, which can be high in some cases. We further build an adaptive scheme that depends only on a *local* version of the growth rate. As a result, the bad areas of the network do not force high resource usage everywhere. We also describe techniques to make peer-to-peer systems tolerant to faults.

Acknowledgements

Most students only get one advisor. I've been lucky enough to have two, John Kubiatowicz and Satish Rao. Hearing two points of view has been incredibly valuable.

Satish, in addition to being a brilliant researcher, is incredibly supportive. He makes me feel like I can do whatever I set my mind to do. I've come to agree with him. Attempting to live up to his opinion made me do things I wouldn't have otherwise.

Kubi has vision that has driven me to expand mine. Talking with him has helped focus my thoughts, and working with him on technical issues is a pleasure. His advice has been instrumental in helping me give better talks and be a better member of the research community.

I would not be here, writing this, if it were not for them. They have been critically important in teaching me how to do research, how to write papers, and how to give talks. The description above does not do them justice. Suffice it to say if you're a student who has a chance to work with either John Kubiatowicz or Satish Rao, take it.

Also, thanks to my co-authors on the work presented in this thesis including (in addition to my advisors) Robert Krauthgamer, Jeremy Stribling, and Ben Zhao. Though I've never written a paper with him, Anthony Joseph's influence has been important. Likewise, Sean Rhea has been an invaluable resource. He approaches some of the same topics as me from a radically different perspective. (He's also the creator of the often-used Figures 2.2 and 2.3.)

Also, thanks to Christos Papadimitriou and Joe Hellerstein, who helped me during my first years in graduate school until I found a topic that suited me.

Ziv Bar-Yossef, my former officemate, has always been encouraging when I needed it most.

I also owe much to both the OceanStore group and the theory group. In the

Oceanstore group, I particular want to thank Ling Huang, Sean Rhea, Jeremy Stribling, and Ben Zhao who wrote the Tapestry code. The interactions with both groups have had important contributions to my research and been simply fun as well.

There are many more people that have impacted my graduate school career in subtle ways. For example, I learned a great deal writing scribe notes for Alistair Sinclair's classes—his comments probably had a more impact than I realized on how I write proofs.

Thanks to Bab5, the Thursday night dinner group who keep me sane through graduate school by connecting me to people outside Soda and Brewed Awaking.

And finally, thanks to John Weale. He's been supportive in big ways and in small throughout graduate school. I'm extremely lucky to have married him.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A peer-to-peer object location system is an evolving set of computers cooperating to store objects. A reasonable system should easily adapt when computers join or leave the network, reliably find objects, and ensure that no computer works too hard. We call these three properties self-organization, completeness, and load-balance. We show that even in such a network, it is possible to find nearby objects. An "object" could be a file, a service, or a particular user. This ability is an important primitive for other applications.

Why peer-to-peer?: As computing devices become smaller and cheaper, there are more of them in our lives, and the increasing number has made them difficult to manage. One challenge for the future is making these small, scattered pieces work together. Peer-to-peer networks are one instance of this problem. The peer-to-peer networks considered in this thesis consist of computers (ordinary desktop machines) connecting together via the Internet. There are interesting issues involved when they are mobile devices, or when the computers have low or intermittent connectively, but we not discuss them here. Applications like OceanStore, CFS, PAST, Fariste, Ivy, and Pangaea [REG$^+$03, DKK$^+$01, RD01b, ABC$^+$02, MMGC02, SKKM02] seek to turn peer-to-peer networks into file systems. PIER [HHB$^+$03] uses a peer-to-peer network as a database. One key primitive of peer-to-peer networks is finding objects (files, users, or services) in the network.

We start with a slightly more formal definition of the problem. Given a set of computers connected via the Internet (or any underlying routing network),

Goal 1 (Object Location). *A set of* publishers *choose locations on which to place objects. A single object could be placed in many locations.* Searchers *look for objects in the network. An* object location *data structure gives publish and search algorithms.*

In other words, we view the network as a distributed data structure that allows the following two operations: *put(key,object,location)* (for the publishers) and *get(key)* (for the searchers). All objects published with the same key are assumed to be the same, so *get(key)* starts by routing a message toward *any* object with that particular key. The *location* argument to the *put* operation is implicitly "here", wherever the *put* operation is invoked. The system should also have low *stretch*; that is, the ratio of the distance traveled to retrieving a copy of the object to the cost of retrieving the closet copy. A low-stretch object location data structure can take advantage of a publisher that places objects near the searchers.

The keys for objects are typically hashes of the object or a hash of the object's human-readable name. Because it is a data structure distributed over the wide area, an algorithm for search or publish is really a series of messages in a network, and so might be more accurately called a network protocol than an algorithm. We use the term "algorithm" and "data structure" rather than "protocol" and "network" because we view the participants not as independent agents, but as pieces of a data structure. Dabek et al. [DZD$^+$03] use the term DOLR (short for *Distributed Object Location and Routing*) to emphasize the important role of routing. Requiring low stretch means that not only is the object copy found nearby (where "nearby" is relative to the distance to the closest copy), the path to get it is also short. This means that the object location data structure can get almost the same advantages as a system that stored the location of the closest copy of the object at each point in the network.

1.1 The fundamentals: self-organization, completeness, and load-balance

Peer-to-peer networks consist of computers linked together via the Internet. This means that any computer in the network could have a connection to any other computer. (This is can be compared to "radio networks", where nodes can only send messages to computers within radio range.)

load balance: Because the peers are ordinary computers, the load on any individual machine must be low. In a network with tens of thousand of participants, keeping links to all of them is impractical. In highly dynamic networks, maintaining links to even a hundred nodes may be prohibitively costly. Instead, each peer chooses some small number of peers to be its *neighbors*, and only communicates with the rest of the network through these neighbors. The set of links naturally forms a graph on the peers.

self organization:A peer-to-peer network must *self organize*. Each node follows some simple protocol to make a decision based on limited information. Any property emerges via local and independent decisions, as no node sees the network as a whole. For example, each peer must be able to choose its neighbors by sending only a few messages. (See Figure 1.1.)

Because these networks are not static, a self organizing network repairs itself. The network should have the ability to tolerate faults for as long as it takes to repair. Repair can be either proactive, or part of maintenance. The proactive approach fixes the network when any change (such as a node arrival or departure) occurs. The other approach is periodic maintenance of the data structure, which requires enough ability to tolerate faults so that waiting to repair does not affect performance or reliably. This thesis takes the first approach, as does [RD01a, RFH+01], and the original Chord paper [SMK+01]. Later work on Chord [LNBK02b, LNBK02a, SMLN+03] and recently Bamboo [RGRK, RGRK03] takes the latter approach. This difference and what it means is discussed in more detail in Chapter 3.

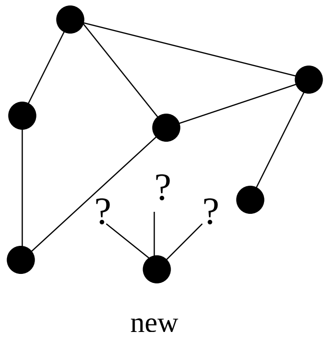

Figure 1.1: *The peer-to-peer world.* When a new node (peer) enters the network, it needs to determine which nodes it should link to.

Part of being self organizing is tolerating the random failures that occur as peers leave the network. Some may leave gracefully, but many just stop responding to messages, perhaps because a link goes down or because a power cord is pulled. Building a network that continues to work under high turnover is rather difficult, and a number of researchers have evaluated exactly this property [LSG$^+$04, RGRK, RGRK03, LNBK02a, LNBK02b] using varying techniques. A more difficult problem is dealing with participants who do not follow the protocol. These can either be malicious, selfish, or merely good participants running buggy code. This issue continues to be a challenge for peer-to-peer networks. Chapter 5 of this thesis addresses a limited class of faults.

completeness: Gnutella is also used to locate objects in a network. Gnutella and related systems do object location by building an unstructured graph on the peers. When a user wants a file, the user asks its neighbors to ask their neighbors, and so on for a limited number of steps, hoping that in that the request reaches a peer with the file. This is called flooding.

The performance of this algorithm depends on the graph formed by the peers in the network. For peer A to find an object on peer B, there must be a path from A to B. Furthermore, there should be few peers on the path connecting A and B. Thus, the ideal graph has low diameter. Random graphs have low diameter, and that is inspiration for networks like Gnutella. Pandurangan, Raghavan, and Upfal [PRU01] give a provably good technique to build networks with short paths ($O(\log n)$) and constant degree even in a changing network.

But even with a good underlying network, flooding the entire network is too expensive, so the search is limited. As a result, the searches often miss objects that are in the network. Uncommon objects are particularly difficult to find.

This maybe-you-find-it-maybe-not is okay for sharing music files, but is not acceptable in other possible applications, such as a file systems (e.g., [REG$^+$03]) , backup (e.g., [WWE$^+$01]) or databases (e.g., [HHB$^+$03]). For these applications, the system needs to find objects more reliably. We call a system that does this *complete.* We design an object location data structure that works perfectly in a perfect network (no lost messages or sudden departures of peers). Further, we show how to add redundancy to deals with imperfect networks. This is described in Chapter 5.

We restate the the basic requirements for an object location structure in a peer-to-peer network.

- **self organizing:** Allowing participants (peers) to enter and leave. Ideally, a self-organizing network works even in the presence of participants that do not follow the protocol.

- **complete:** If the network does not change, objects in the network are found during a search. That is, once a *put(key,object,location)* completes, a *get(key)* returns the object.

- **load balanced:** No participant should be overloaded by network traffic or storage requirements. Part of this means requirements (in terms of storage and network connections) should grow no faster than $O(\log n)$, where n is the size of the network.

While others have achieved the above goals, this thesis discusses a solution that achieves these properties and low stretch, which we motivate below.

1.2 Low Stretch: Efficient Object Location

A good object location algorithm should also find objects quickly. For file-system applications, waiting for a round trip across the Atlantic for every object access is impractical. In other words, lookups should be local when possible. For an item stored within Berkeley, a Berkeley searcher should not have to send messages to MIT. To measure the locality of the object location data structure, we use *stretch*. Stretch is the ratio of the distance traveled as part of the search request to the distance to the nearest copy of the object. (For this thesis, distance is network latency. We assume distances are symmetric and satisfy the triangle inequality. When we write "the metric space" we are talking about the metric space defined by the pairwise latency measurements. A graph also gives a metric space.)

Why stretch?: Applications like file systems [REG$^+$03, ABC$^+$02, DKK$^+$01, MMGC02, SKKM02] are too slow to use if every access must go across the wide area. Any practical peer-to-peer file system must make use of local resources.

To avoid crossing the network, copies of objects must be placed close to their access points. Systems like Chord [SMLN$^+$03] and Pastry [RD01a] hope that by placing enough copies in essentially random locations, one of the copies is close. This could require placing many, many random copies when a clever publisher need only place a few to get equal performance. For example, a clever publisher places the the Berkeley room reservation list in Berkeley, and places the MIT phone directory in MIT. On the other hand, if copies are placed at random (according to ID), many copies would have to exist before there was one in Berkeley. Most of those copies would be a waste of both space and update traffic. This thesis does not address placement schemes, though there is much work in the area (*e.g.* [RRRA99, KR03, DW01, CKK02, LGI$^+$99, KRS00, CCW$^+$]). Instead, our goal is locate objects (finding the nearby copies, when they exist) without sacrificing the peer-to-peer requirements of self-organization, load balance, or completeness. In doing so, we enable clever placement algorithms.

Minimizing stretch may also give two other benefits besides reducing latency. First, it may reduce bandwidth usage. Suppose a dorm user is looking for a popular song. That song is probably available elsewhere in the dorm, and sending a message outside the dorm looking for the song and downloading it from another university is an expensive use of bandwidth. A perfect low-stretch lookup scheme ensures that object requests do not use any links outside the local area if the object is within the local area. Second, low stretch means that requests are kept local much of the time. This can be very important under network failures. It means that if a piece of the network gets isolated, many requests may not even notice.

Notice that when objects are placed randomly in the network, getting low average stretch is relatively easy, since almost all objects will be far away. So, this dissertation assumes that objects have been placed at *arbitrary*, and not random, locations.

For the analysis, the publishers are "adversaries," [1] and the goal of our algorithm is to get low stretch even when the publishers place objects in the worst case location for the data structure. It turns out that the worst-case locations are locations *nearby* the searchers, since the algorithm must be very efficient in this case.

Routing Stretch: A note is needed on the term stretch. Our definition of stretch is more precisely called *object location stretch* to distinguish it from *routing stretch.* Routing stretch is the ratio of the length of the overlay path between two participants to direct path in the Internet. Thus, the technical difference is that routing stretch is between two specific endpoints, while object location stretch measures the distance traveled to find a copy (any copy) of an object over the distance to the closest copy. More importantly, the two terms are used in different contexts to measure different things. Routing stretch is typically is given over the whole network (by presenting the mean, or the median, or the 90th percentile), and thus effectively measures the inefficiency of the routing when the the pairs are chosen at random. On the other hand, as we described above, object location stretch is a worst-case measure. A

[1] The publisher, however, is an oblivious adversary and does not see the random coin flips that determine the node IDs.

network with low routing stretch could easily have a very high object location stretch. (See Appendix A for an example.)

1.2.1 Simple, but inadequate, solutions

To make the above discussion more concrete, this section gives two simple object location data structures and shows why they are not practical solutions in the peer-to-peer world.

Directory assistance: One simple solution is to keep a central directory. (This is essentially the Napster [Fan] solution.) A peer is chosen to be the directory. Publish (or *put*) requests put a *key,location* pair on the directory, and search requests visit the directory looking for the location of the object with a particular key. This is analogous to calling 411 to find a person's phone number. However, this solution is neither low stretch nor load balanced.

To see why the stretch will be high, notice that if the directory is located in North America, users from Australia will be at a disadvantage, since every lookup will have to go to North America. Not only does this mean slow service for users in Australia, but it also potentially stresses the links between Australia and the North America, since even requests from Australia for objects in Australia travel to the North America.

Second, the central directory has an unacceptably high share of the work. Outside the peer-to-peer world (i.e., for Napster), it may be possible to provision a single computer (or a cluster of computers) to handle the load. But the participants in the peer-to-peer network cannot be assumed to have this capability, so it is deadly to overload a single machine. The owner whose desktop machine has become the central directory is likely to be unhappy about that, since it makes the machine unusable. Moreover, for some problems, it may be too expensive or outright impossible to build a centralized server or cluster. If the directory fails, either by accident or design, the whole network fails. Thus, it is extremely important to maintain the availability of the directory.

Phone Book: Another simple solution is for each participant to keep a list of all objects and where they are located. This is like distributing a phone book to all the participants. Notice that once the phone book or object list has been distributed, lookups are easy, but updates are difficult. Lookups for objects go directly to the objects (so lookups in Australia stay in Australia), so the stretch is one. On the other hand, when an object is added or deleted, all the peers in the network must be contacted. As a result, each peer stores a lot of information, and even worse, the communication for each *put* message is high.

If updates are rare, and a small amount of inconsistency can be tolerated (perhaps there is a slower but reliable backup method of performing a *get*), this may practical. Gupta, Liskov and Rodrigues [GLR03] present a peer-to-peer network that uses essentially the phone book idea. They calculate the relationship between the node turnover (arrivals and departures) and update bandwidth, and show that for reasonable network sizes and reasonable turnover rates, this solution is practical.

1.2.2 Hierarchical Object Lookup

The ideal is the stretch of a phone book solution, but with less update traffic and lower storage. The natural way to do this is to use a hierarchy of directories. Nearby objects are found in local directories that are easy to access, while far away objects are found in more distant directories (but since the objects were far away, the stretch is low, even in this case). That is, the Australians check the Australia directory before visiting the North American directory.

To use a non-mathematical example, peer A contacts the city directory to see whether the object is in the city; if not, it tries the state directory, if not there, it tries the country directory, and if not there, it visits the world directory. If the cost of accessing the DHTs grows geometrically (i.e, the cost of a city-wide search is a constant less than the cost of a state-wide search), then the total cost will be a constant times the cost of the level where the item is actually found.

Even in this metaphor, three pitfalls emerge.

- Self-organization. A node entering the peer-to-peer network does not start out with a clear notion of where it is located. It terms of the city/state/country analogy, a peer entering the network does not enter knowing its city, state or country.

- Boundary cases. Consider a peer looking for an item that is just across the state line. The item is not found in either the state or city directory, even though it is physically quite close.

- Load balance. The directories (particularly the global directory) suffer the same problems as the central directory.

Two levels of the hierarchical solutions are represented in Figure 1.2. On the left, the black nodes are local leaders at some level of the hierarchy, and divide up the space (as shown by the dotted lines). On the right, again, the black nodes represent local leaders of a bigger regions, divided by the the solid lines. If two nodes are close to each other, they are likely to belong to the same region.

The problem of self organization is choosing these centers such that they are the right distance apart and ensuring that nodes entering the network are able to find their nearby center. The boundary problem is visible here, too, in that some nodes close together are separated by lines and so in different regions. (Appendix A gives a thought experiment showing this affect.) Notice, also, that the higher up in the hierarchy, the fewer neighbors are separated. The load balancing problem is that the black nodes act as central directories for their regions, and so do an unfair share of the work, particularly at the higher levels where the regions are larger.

Stretch and space: There is a natural trade off between stretch and space. For each level of the hierarchy, the publisher places key-location pairs in the regional directory. The more levels of the hierarchy, the more key-location pairs. But with

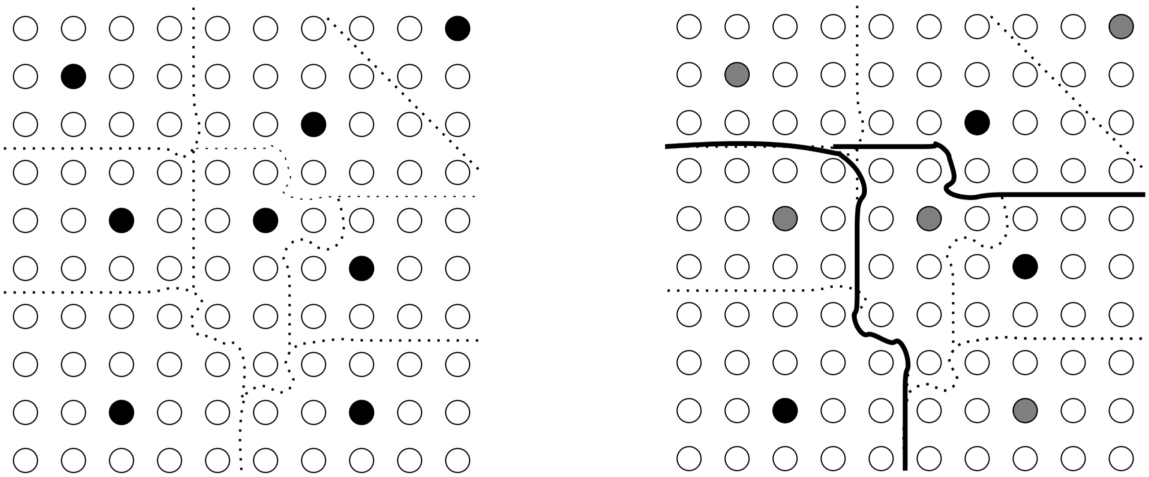

Figure 1.2: *Building a hierarchy.* The set of nodes black nodes partition the grid. The left shows one level of the hierarchy, the right the next higher level. Two nodes that are close to each other are most often—but not always—in the same region.

only a few levels of the hierarchy, the stretch is high, since traveling to the directories is more expensive. For example, if we eliminate the city-level directories, then objects in the city are not found until the state-wide search, and since the state-wide directory is far away, this increases the stretch. The central directory, as hierarchy with only one level, is the extreme of the low-space high-stretch. At the other end of the space-stretch spectrum is the phone book solution, with very high space, and very low stretch. Section 1.2.4 gives a number of solutions with moderate space and moderate stretch.

The low stretch object location system we present uses a hierarchical approach. The overall structure is described in Chapter 2, where we revisit this picture. The key idea is to use random choices of node IDs to do load-balancing and structure the hierarchal. The key pieces to self-organization are described in Chapter 3 and Chapter 4.

1.2.3 DHTs: self-organizing, complete, load-balanced, but high stretch

Distributed Hash Tables, or DHTs for short, address the overloading problem inherent in a centralized directory. However, they do not decrease the stretch over the central directory solution, and may even increase it, making them an incomplete solution for object location.

A DHT views the peers in the networks as buckets in a hash table. The DHT interface is like a hash table, with two operations: *put(key,object)* to put key, object pairs into the network, and *get(key)* to retrieve the object associated with the given key. Notice that there is no *location* argument to put. This is the main difference between DHTs and object location data structures, and means that DHTs have less information to work with. Traditional hashing would choose a bucket by taking the ID of the object modulo the number of peers in the network, but this would be a terrible thing to do in the peer-to-peer scenario because the number of buckets is constantly changing. Instead, DHTs use some variant of consistent hashing, first described by Karger et al [KLL$^+$97]. With this technique, object IDs and node IDs (also chosen via hashing so they are random) are in the same namespace, and objects are mapped to the node with the most closely matching ID. As a result, when a peer enters or leaves, only the objects assigned to that peer must move.

Storing a list of all peers would be quite expensive in a large network, and most DHT proposals seek to limit the numbers that needs to be directly stored. (Again, [GLR03] is an exception.) The DHT proposals a network on these peers that allows for easy routing to a specified ID. There were essentially three simultaneous proposals: CAN [RFH$^+$01], which arranges the buckets in a grid, Chord [SMLN$^+$03] which arranges the buckets in something like a butterfly network,[2] and Pastry [RD01a]. (The first Tapestry [ZHR$^+$03] proposal appeared at this time, but it is an object lo-

[2] Chord is often viewed as a ring, which is a more useful viewpoint when considering fault-tolerance. But the ring structure alone without the finger links would be completely unsuited to a peer-to-peer network.

cation system, though it can also serve as a DHT.) Pastry (like Tapestry) is based on a non-dynamic scheme of Plaxton, Rajaraman and Richa [PRR97] that resembles a hypercube.

In a typical DHT construction, every node at most $O(\log n)$ neighbors, and finding a particular key requires only $O(\log n)$ steps. All also give distributed algorithms for node join and leave.

But DHTs suffer from the same problem as the central directory; the stretch is high. The *put* puts the key-object pair on a random node in the network. Thus, any request for an object must always travel to a random peer. An object created in Stanford and accessed from Stanford may have its key stored in the UK, and then every access must first go to the UK to find where in Stanford the object is stored. Recent work on DHTs has optimized for locality, but at best, DHTs will have the same performance as a central directory—when the peer hosting a given key is far away, the DHT *must* send a message to that location. When objects are placed at random, the peer hosting the key is probably not that much further than the peer containing the object, so a DHT works relatively well. However, when objects are not placed at random, or are available in many places, the stretch is very high.

DHTs are self-organizing, complete, and load-balanced. Thus, they meet the peer-to-peer requirements, but they cannot be made low stretch.

1.2.4 Low-stretch networks

A variety of problems involve building low-stretch networks, even before peer-to-peer networks existed. One such problem is routing with name independence. Imagine that the nodes in a network are given arbitrary names. The goal in routing is to build tables so that a message addressed to node A can get there.

An "object" is much like a node whose name is arbitrary (*i.e.* not dependent on location), so searching for an object is essentially the same as routing to a node when node names are independent of location. There are a couple of differences, however.

Name independence

Destination	**Direction**
Times Square	West
Chrysler Building	East
Empire State Building	South
Central Park	North
⋮	⋮

Name dependence

Destination	**Direction**
Avenue < 5	East
Avenue > 5	West
Street < 42	South
Street > 42	North

Figure 1.3: *Routing in Manhattan.* The left shows a piece of the name-independent table for 42nd and 5th in Manhattan. The right shows the *complete* name-dependent routing table for same intersection.

First, there may be multiple copies of the same object. Second, there may be many more objects than nodes, so using $O(\log n)$ storage per node might be reasonable, while paying $O(\log n)$ per object may not be. However, these systems are typically structured such that there need only be one routing table per physical location thus the only cost of adding an object to the network is adding a few pointers in the right places. As a result, object location and name independent routing are essentially the same problem.

To get a stretch of 1, routing takes place along shortest paths, and every node in the network can maintain a table that for each node name, lists the outlink to be used to get there. Consider the network defined by the streets of Manhattan, and the intersections are the nodes or peers. The routing table is a mapping between destinations and direction. (See Figure 1.3, left) Notice even though there are only four possible directions, this table could be as large as the total number of destinations! If the destination are described by cross street and not by name (making name dependent on location), the table only has four entries *total.*

In the general case, it is impossible to use much smaller tables and still get a stretch of one even when names are allowed to depend on location [GG01, GP96]. (Manhattan, since it is more or less a grid, is a special case.) But if the paths are allowed to be slightly longer, the size of the tables can be reduced. Thus, there is a trade-off between stretch and space. A directory at ever node uses space of $O(n)$ per node, and gives $O(1)$ stretch. Routing in a spanning tree gives $O(n)$ stretch, with space can be $O(1)$.

A good name-independent routing scheme finds an optimal or near-optimal point in the space-stretch trade off. Awerbuch et al. [ABNLP89] introduce the first non-trivial name-independent scheme. Awerbuch and Peleg refine this technique [AP90, AP91]. Their system is complete, but neither load-balanced nor self-organizing. Later, Rajaraman et al [RRVV01] build a system for general metric spaces based on hierarchically well-separated trees that is load-balanced and complete, but not self-organizing.

Other recent work on name-independent routing includes [ACL+03b, ACL03a], both of which gets close to the optimal trade off (the first is for networks with only bidirectional links, the second for networks with one-way links). Abraham et al [AGM+04] improved on this, showed a stretch of 3 is possible with $O(\sqrt{n})$ space (ignoring logarithmic factors). Though these schemes achieve a space-stretch trade off that is close to optimal, the space usage is perhaps too high to be usable in peer-to-peer networks.

Also related is the work on compact distance oracles by Thorup and Zwick [TZ01]. Their scheme gives as a byproduct an object location algorithm (though the result isn't load balanced).

Recently, Krauthgamer and Lee [KL04b] gave a nearest neighbor search structure that combined with publish techniques used in [PRR97], yields an object location scheme that is self-organizing and complete, but not load-balanced. The scheme has $O(1)$ stretch in a restricted class of metric spaces. Table 1.1 summarizes these results.

Low Stretch Object Location and Hierarchical Lookup: These schemes use some variant on hierarchical search. One of the first such schemes was that of Awerbuch and Peleg [AP91]. They build a hierarchy with $\log D$ levels, where D is the largest distance between two nodes in the underlying network. For the ith level of the hierarchy, ideally [AP91] divides the graph into regions of diameter 2^i. Then, for each i, the publisher puts a *key-location* pair in its ith level directory. The searcher looks for an object by checking first the $i = 0$ directory, then then $i = 1$, and so on. Suppose the searcher finds the object in the kth directory. Then the total cost to the searcher is $1 + 2 + 4 + \cdots + 2^k = 2 \cdot 2^k$. Thus, if the object was about 2^k away, the stretch is constant. However, this does not quite work because of the boundary problem: two nearby nodes could end up in different regions (and it could happen at many levels of the hierarchy) As a result, [AP91] expand the regions such that they overlap. This causes a couple of problems. First, the more regions are expanded, the more expensive it becomes to reach the regional directory, increasing the stretch. Second, no participant can be in too many regions. Another paper by the same authors [AP90] gives a way to construct a set of clusters that trades off between these two requirements. In particular, for each i, they show a way to give a set of clusters such that

- Every pair of nodes within distance 2^i share a cluster.
- No cluster has diameter more than $O(2^i \log n)$.
- No node is more than $O(\log n)$ clusters.[3]

Using these regions, they build an object location system with $O(\log n)$ stretch.

RRVV [RRVV01] also use hierarchical lookup. They build their clustering using hierarchical well-separated trees [Bar96]. Within each cluster, [RRVV01] balances the load using essentially a peer-to-peer network based on de Bruijn graphs. Their scheme is not self organizing.

Thorup and Zwick [TZ01] use randomization to build clusters for creating compact distance oracles. This scheme is easy to modify to an object location scheme. For each level, their scheme randomly picks a subset of the nodes to be cluster leaders. Each node then chooses the closet cluster leader to be its leader. The idea, again,

[3] The result is slightly more general. The last two items trade off, such that no cluster has diameter more than $O(2^i k)$, and no node is in more than $O(kn^{1/k})$ clusters.

is that two nearby nodes are likely to have the same cluster leader. Because of the boundary problem, this does not happen, so nodes also store a few other nearby cluster leaders. With this additional information, nearby nodes will share a cluster leader. As before, [TZ01] is not self organizing or load balanced.

Krauthgamer and Lee [KL04b] also build a hierarchical set of clusters. The paper makes assumptions about the underlying network, and so they are able to guarantee find a hierarchical set of clusters with better properties than [AP90]. In particular, the diameter of a clusters in [KL04b] is only a small fixed constant factor larger than 2^i, and each node is only in a small number of clusters, where that number is a constant depending on the network. Their scheme is self organizing, low stretch (in special metric spaces), but not load-balanced.

The compact routing schemes [ACL$^+$03b, ACL03a, AGM$^+$04]) achieve extremely low stretch (5 and 7 in the first, 6 for directed networks in the second, and 3 in the last) using a two-level hierarchy. At the first level, every participant stores the outlink (i.e. north,south,east or west) for each of the closest $\sqrt{n}$ nodes. Each participant also stores the directions to $\sqrt{n}$ directories, each of which is in charge of $n/\sqrt{n} = \sqrt{n}$ objects. Ignoring logarithmic factors, each node ends up storing about $\sqrt{n}$ information, and so the scheme is load balanced. However, the total storage is too high for the peer-to-peer scenario. The first two [ACL$^+$03b, ACL03a] also introduce a scheme that uses less storage by adding lookup levels to the hierarchy, but with k levels, the stretch grows too fast for the the peer-to-peer case. None of the three schemes are self-organizing, because they do not give an efficient way of finding the closest $\sqrt{n}$ nodes.

One key point is that self organizing is blocked in several schemes by the need to find the nearest member of some set. The only self-organizing low-stretch scheme mentioned here [KL04b] is a nearest neighbor search structure designed for a space in which nearest neighbor search is efficient. Thus, the key barrier to building self-organizing low stretch systems may be finding the closest from some set.

System	**self-organizing**	**complete**	**load balanced**	**low stretch**
Phonebook	no	yes	yes	yes
Directory	no	yes	no	no
DHTs	yes	yes	yes	no
Gnutella	yes	no	yes	no
Awerbuch-Peleg [AP90]	no	yes	no	yes
RRVV [RRVV01]	no	yes	yes	yes
PRR [PRR97]	no	yes	yes	sometimes
Krauthgamer-Lee (implicit) [KL04b]	yes	yes	no	sometimes
Compact Routing [ACL+03b, ACL03a, AGM+04]	no	yes	yes, but big	yes
Thorup and Zwick [TZ01]	no	yes	no	yes

Table 1.1: *A comparison of object location techniques..* A "sometimes" in the low stretch column means that the scheme is low stretch for special metrics.

1.3 Our Results

This thesis shows a low-stretch object location data structure for peer-to-peer networks. That is, the scheme presented is be self organizing, complete, load balanced and also low stretch. We use as a starting point the low-stretch scheme of Plaxton, Rajaraman, and Richa [PRR97]. This schemes assigns random IDs to the nodes to get load balanced hierarchical scheme that guarantees constant stretch in a certain class of networks. The main flaw in PRR scheme is that is cannot self-organize. Our work give self-organization algorithms for it. In the Chapter 6, we expand the range of networks for which the scheme applies. A chapter-by-chapter description of this thesis follows.

Chapter 2 Background.: This chapter describes the basic scheme for unchanging networks. It explains the prefix routing scheme of PRR [PRR97] which is the starting point for Tapestry, the systems for which these algorithms were developed and implemented. It is a hierarchical scheme that uses randomization to build the hierarchy. It also uses the randomization to load balance. If the network meets the growth-restriction requirement (roughly, that the network is grid-like from some dimension d), the increase is geometric, so the length of a route is always dominated by the last hop. This chapter also explains how the publish and search algorithms use the underlying routing structure is used for low-stretch object location in these growth-restricted spaces. We end the chapter by showing exactly how our scheme deals with the three challenges for a hierarchical scheme.

Chapter 3 Node Join and Leave: Developing algorithms to allow nodes to join the data structure has two parts. This chapter covers the algorithms needed to maintain the routing structure, which is necessary for reliable object location. The second part, ensuring that the stretch is low, is covered in the next chapter. The chapter starts by giving algorithms for the the case of a single join, performed in isolation. This involves determining the set of peers that must know of the new peer and notifying them of the new node's arrival. Section 3.2 extends the algorithm to work with multiple, simultaneous joins. The challenge here is to ensure first that two nodes joining at the same time find out about each other, and we show how to do it in a lock-free fashion. We then address node departure.

Chapter 4 Nearest-Neighbor: The routing structure described in Chapter 2 requires that each node be able to find the nearest node whose ID matches some prefix. In order to maintain this in the presence of joins and leaves, we need an algorithm for finding the nearest neighbor in the network. This section uses the tree defined by the set of paths to a particular destination backward as nearest neighbor search structure. Thus, nearest neighbor search requires no additional storage over the routing structures.

Chapter 5 Fault Tolerance: This chapter shows how to make the structure tolerant to faults. First, it looks at the problem of routing to a destination. If nodes along the routing path are faulty, then messages may not reach the destination. Instead of ensuring that there are multiple paths between source and sink, it creates a "wide path", in essence checking progress between each step. This technique delivers messages with high probability using $O(\log n)$ redundancy. If the network is growth-restricted, then the technique works even against a malicious adversary of limited power.

Second, chapter looks at increasing the tolerance of the nearest neighbor algorithm to faults. The algorithm of Chapter 4 could fail if even one node is faulty; this chapter shows that adding O(log n) redundancy to the algorithm ensures that it finds the correct nearest neighbor with high probably even when a constant fraction of the nodes have failed. The chapter also presents simulation results showing that that algorithms make a significant difference in realistic networks with a factor of 3 redundancy.

Chapter 6 Generalizing the Metric: This chapter draws the connections to other problems. First, it proves, using a technique similar to that used by Bourgain [Bou85] and Linial, London, and Rabinovich [LLR94], that a modification of the scheme described in Chapter 2 can be used for object location in general metric spaces (this originally appeared in [HKRZ02]). This construction isn't useful in the peer-to-peer arena because it cannot be easily load-balanced, leading to an interesting open problem.

The main contribution (which appeared first in [HKK04]) is to expand the set of metric spaces for which there exists an object location scheme. In doing so, it seems to push up against the lower bounds for nearest neighbor search proved by [KL04a], suggesting a connection between object location and nearest neighbor search.

Chapter 7 Conclusions and Future Work: The final chapter gives the lessons learned and presents some related open problems.

Chapter 2

PRR-trees and Tapestry

In this chapter, we describe PRR-trees, first introduced in by Plaxton, Rajaraman, and Richa [PRR97]. We describe a particular implementation, called Tapestry written at Berkeley. Tapestry is first described in [ZKJ01], and is a practical object location structure based on [PRR97]. In the process, we show DHTs and object location systems are indeed closely related.

PRR-trees use hierarchical lookup. They select a random subset of nodes to be local directories, and then a random subset of those nodes are one level up directories, and so on. The key element of their solution is to use random ID assignments to select the set of local directories. Thus, if roughly $1/b$ nodes should be local directories, they assign nodes random names in base b. Then, all the nodes with the first digit of the ID 0 are local directories. This approach allows for load balancing, as follows. Each $j \in [0, b-1]$ defines a partition by letting the nodes with first digit j form the set of local directories ("city halls" to continue the analogy from the introduction) and associating each node with its closest local directory. Thus, instead of one set of local directories, there are b sets. Which set is used for a given object depends on the object's random ID, and thus, objects are randomly assigned to a directory set. This continues at every level of the hierarchy.

We now switch viewpoints and present PRR trees as prefix routing networks Tapestry divides naturally into two pieces, a routing layer and an object location layer. The routing layer can be used by itself as a DHT, and in fact, Pastry [RD01a] and Bamboo [RGRK, RGRK03] (both DHTs) use essentially the same routing layer as Tapestry.[1]

Nodes (or peers) in Tapestry are given IDs. We assume these IDs are random; in practice, these IDs are hashes of the node's IP address or some other data. These IDs act as the keys used in the *put* and *get* commands mentioned earlier. Objects will also be given IDs, and as with nodes, we get these IDs by hashing either the object itself, or a human-readable name for the object. We will refer to node identifiers as *node-IDs* and object identifiers as IDs. We assume these random IDs are long enough that no two of them are the same.

Section 2.1 describes how PRR-trees (of which Tapestry is one) are constructed so that routing to a given ID is an easy operation. Section 2.1.1 then describes what is done when the desired ID does not exist, comparing the Tapestry solution to that of PRR and Pastry. Section 2.3 describes how to use this routing structure for object location, again comparing Tapestry to PRR. Finally, Section 2.4 explains how Tapestry and PRR overcome the three challenges inherent in hierarchical object location systems.

2.1 Routing Network

This section builds a network that gives efficient routing to a given ID. For this purpose, consider the ID string as a string of digits in radix b. For a string of digits α (Greek letters will usually denote ID strings), let $|\alpha|$ be the number of digits in that string. Tapestry inherits its basic structure from the data location scheme of Plaxton, Rajaraman, and Richa (PRR) [PRR97]. As with the PRR scheme, each Tapestry node contains pointers to other nodes called *neighbor links*.

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

[1] Tapestry and Pastry were developed at approximately the same time. Bamboo is a later system.

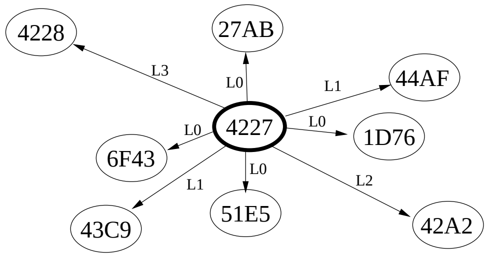

Figure 2.1: *Tapestry Routing Mesh.* Each node is linked to other nodes via *neighbor links*, shown as solid arrows with labels. Labels denote how many digits are shared between the two nodes. Here, node `4227` has an level 0 link to `27AB`, resolving the first digit, a level one link to `44AF`, resolving the second digit, etc. Using the notation of Section 2.1, 42A2 is a $(\mathtt{42}, \mathtt{A})$ neighbor of `4227`.

with α. Neighbor links are labeled by their *level number*, which is the length of the shared prefix. Figure 2.1 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 node A stores R nodes with prefix $\beta \circ j$ (preferably, these are the *closest* nodes with that prefix). This set of neighbors is denoted $\mathcal{N}^A_{\beta,j}$. If the number of nodes with prefix $\beta \circ j$ is less than R, A stores all such nodes.

For each j and β, the closest node in $\mathcal{N}^A_{\beta,j}$ is called the primary neighbor, and the other neighbors are called secondary neighbors. When context is obvious, we will drop the superscript A. Let $l = |\beta|$. Then, the collection of b sets, $\mathcal{N}^A_{\beta,j}$, form the level-l routing table. There is a routing table at each level, up to the maximum length of node-IDs. This gives the following important property.

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 prefix $\beta \circ j$.*

Property 1 implies that the routing mesh is fully connected. Messages can route from any node to any other node by resolving the destination node-ID one digit at a time. Let the source node be A_0 and destination node be B, with a node-ID equal to $\beta \equiv j_1 \circ j_2 \ldots j_n$. If ϵ is the empty string, then routing proceeds by choosing a succession of nodes: $A_1 \in \mathcal{N}^{A_0}_{\epsilon,j_1}$ (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 that is used in the object location scheme.

Efficient O(diameter) **routing:** If the underlying network is structured nicely, then this construction means that the length of the ith hop is about δ^i for some δ, and combined with the fact that the longest hop is distance about diameter (or the largest distance in the network), the total distance traveled in routing is O(diameter). (This is not the same as getting low stretch.) The key observation is that the distances to the level-i neighbors is always going to be shorter (or the same length) than the level-$(i + 1)$st neighbors because there are fewer nodes that could be level-$(i + 1)$ neighbors. More concretely, in a two-dimensional grid, increasing the radius by a factor of $\sqrt{2}$ increases the number of nodes by a factor of 2. When the digit size is 2 (*i.e.*, the IDs are in binary), this mean than a level-$(i + 1)$ neighbor will be about $\sqrt{2}$ times further away than the nearest level-i neighbor. (When the digit size is b, the level-$(i + 1)$ neighbor is about $\sqrt{2}^{\log_2 b}$ times further away.) As a result, each hop is a constant factor longer than the previous one, so the total distance traveled is a constant times the last distance, and the last distance is no more than the diameter of the network.

Chapter 6 contains a formal proof of this, as do [PRR97, AMD04].

2.1.1 Tapestry as a DHT: Surrogate Routing

DHTs work by defining a route-to-ID operation that routes to the node with the closest node-ID to the given object-ID. This mapping between object-ID and node-

ID has to be the same at all points in the network, and change as the network changes. For an object with ID ψ, the node with the closest matching ID is $\mathcal{R}_\psi$ and the process of getting to it is *surrogate routing*, so named because it involves routing toward ψ as if it were a node, then adapting when the process fails. Giving a definition of $\mathcal{R}_\psi$ and a way of finding $\mathcal{R}_\psi$ gives a DHT as follows. A *put* request for an object with ID ψ places a key, location pair at $\mathcal{R}_\psi$ (the key is ψ). A *get* request for object ψ retrieves the key, location pair from $\mathcal{R}_\psi$. Note that $\mathcal{R}_\psi$ changes as the network changes, but that at any given moment, a request for object ψ must always go to the same $\mathcal{R}_\psi$.

There are many good ways to choose $\mathcal{R}_\psi$. PRR annotated the routing table with additional neighbors to be used when normal routing failed. 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, but use the links we already have. As a result, all decisions are local, so we call such a scheme localized routing. All routing decisions are made based on the current routing table, the source and destination IDs, and information collected along the route by the query (*e.g.*, the number of digits resolved so far).

We highlight two variants of localized routing; others are certainly possible. We also include the scheme used by Pastry and Bamboo for comparison. The two localized schemes proceed by routing one digit at a time toward the destination ID, that is, each network hop resolves one additional digit toward the destination. Since there is no backtracking, these schemes are guaranteed to complete. Routing stops if the current node is the only node left at the current level in the routing table; the resulting node is the root node.

- **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. The same process happens for the next digit, until there is no other node in the table.

- **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 ID in the most significant bits.

- **Pastry-like Routing:** In Pastry [RD01a] and more recently, Bamboo [RGRK03, RGRK], in addition to the neighbor table, nodes maintain a list of the next k greater IDs and the next k smaller IDs, called a *leaf set.* Then $\mathcal{R}_\psi$ is the node with the closest matching ID to ψ. Routing proceeds as before. When prefix routing fails, routing is done in the leaf set until the destination ID.

The localized schemes do not require any information not already in the neighbor table. On the bad side, it is difficult to describe the $\mathcal{R}_\psi$ without describing the routing process, and it also has the property that a node entering the network may take over objects from many nodes. In comparison, the Pastry scheme is simpler to describe, and a new node entering the Pastry network only takes objects from two other nodes.

However, for almost all purposes, the choice between the Tapestry-style surrogate routing and the Pastry-style surrogate routing is a matter of aesthetics and not functionality. Anything that can be done with one scheme can be done in the other for a nearly equivalent cost.[2]

[2] The exception may be dealing with faults. The Pastry leafsets form a connected graph all by themselves, and as a result, it is possible to route (if only inefficiently), even when the rest of the neighbor table is lost.

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

Proof. The proof is 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. □

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 [ZKJ01].

2.2 Multiple roots

If the root for an item with ID ψ fails, the pointer to the object is lost. This is addressed by in part by using soft-state, so that objects re-announce themselves to the network (republish) ensures that the object eventually reappears. However, in the intervening time, the object is unavailable. Republishing frequently enough to make this interval small enough cause an impractical amount network traffic, so we use backup roots.

For this reason [HKRZ02, HKRZ03] suggested that an item publish itself under several IDs (by hashing the ID with different salt). A search that failed for one ID could try the other IDs. With enough replication, the risk that all roots for an item disappear before it republishes is small.

Rowstron and Druschel [RD01a] also faced this problem. Instead of independent backup roots, they used the leaf set (recall that leaf set is the k nodes with the most closely matching ID) as backups. As a result, rather than k separate *put* messages for k backup roots, only one *put* message is needed, and that nodes puts the object on the other roots. There are two more advantages to this approach. First, if the original root disappears, the next-most-closest node in ID space is a backup root, and so *already* has a pointer to the copy of the object. In the original Tapestry solution, the routing essentially has to restart with a new ID from wherever the old routing path failed. Bamboo also uses a technique like Pastry, and takes it still further by having members of the leaf set periodically contact each other to exchange objects in common. Since roots that share objects know each others IDs and so can exchange information to repair the network without a republish message [RGRK, RGRK03]. While this may be possible with k independent roots, it is not as natural. One advantage of the k independent roots in Tapestry is that they could be used in parallel to get lower latency object requests, and the Pastry leaf sets would not have the same effect.

In retrospect, it seems clear that the Pastry style backup root technique is the better one. However, note that it is *not* the definition of surrogate routing that makes the difference in this case. The key difference is that in the Pastry scheme, the definition of backup roots is connected to the surrogate routing scheme in a tight way. In particular, the backup roots are hot backups, which the routing algorithm naturally finds if the original root goes down. A similar thing could be done with Tapestry's surrogate routing scheme. In particular, the k backup roots for an object with ID ψ are the k nodes matching in the most digits possible. (This doesn't quite end up the same as Pastry's leaf set—consider the all zeros item. The Tapestry root set would only include items with a prefix of zeros, while the Pastry set would include lower items.)

In the rest of the thesis, we give the algorithms that assume a single, good, root, but it is easy to extend them to the case where the "root" is conceptually a set of nodes that are nearby in the ID space. Thus, they may be the Pastry or Bamboo leaf

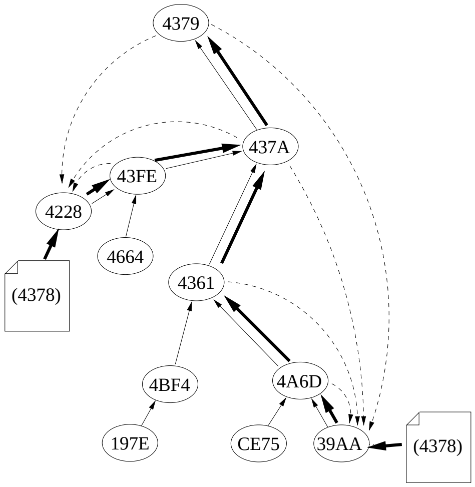

Figure 2.2: *Publication in Tapestry.* To publish object 4378, server 39AA sends publication request toward root, leaving a pointer at each hop. Server 4228 publishes its replica similarly. Since no 4378 node exists, object 4378 is rooted at node 4379.

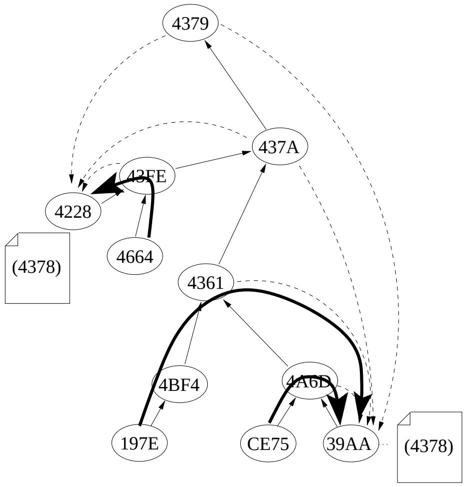

Figure 2.3: *Routing in Tapestry.* Three different location requests. For instance, to locate ID 4378, query source 197E routes toward the root, checking for a pointer at each step. At node 4361, it encounters a pointer to server 39AA.

set or a Tapestry version of the leaf set.

2.3 Object Location

The previous section, gave a network that allows for routing to a given ID. In this section, we show how to use this structure for object location.

Recall that Tapestry maps each object ID, ψ, to a *root* $\mathcal{R}_\psi$ as described in the previous section. Storage servers *publish* the fact that they are storing a replica by routing a publish message toward $\mathcal{R}_\psi$. (Publishing an object with ID ψ is the same as *put(ψ,obj,here)*). Publish messages are routed along primary neighbor links. At each hop, publish messages deposit *object pointers* to the object. (An object pointer is another name for the key-location pair.) Figure 2.2 illustrates publication of two replicas with the same ID. 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 the object moves or is updated.

Searchers for object ψ route toward the root nodes $\mathcal{R}_\psi$ 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). Figure 2.3 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.

It is these intermediate pointers that make Tapestry an object location system. Because of the way the neighbors are chosen and the way that routing works, pointers are nearby the object, and nearby searchers find pointers left by publishers. Consider a nearby publisher and searcher. The publisher has placed pointers to the object on the path toward the object's root. See Figure 2.3. The publisher's (at `39AA`s) first hop is the closest node beginning with 4. The searcher at `CE75` likewise chooses, as its first hop, the closest node beginning in 3. Since the publisher and searcher are very close, this node is the same, and the searcher find the object pointer at that first hop, and then goes directly to the object. The searcher at `197E` is a little further away, but at its second hop (to `4361`) it meets up with the publisher's pointer.

In an arbitrary network, it may be impossible to build an efficient, constant stretch object location system. PRR restrict their attention to metric spaces with a certain even-growth property: they assume that for a given point A, the ratio of the number of points within $2r$ of A and the number of points within distance r of A is bounded above and below by constants. (Unless all points are within $2r$ of A.) Given this constraint, [PRR97] shows the average distance traveled in locating an object is *proportional* to the distance from that object, that is, queries exhibit $O(1)$ stretch. More recently, Abraham, Malkhi and Dobzinski presented a much simpler proof of a similar scheme, using only an upper bound on the size of the number of points within $2r$. Chapter 6 presents a proof of low stretch for networks parametrized by a local, rather than

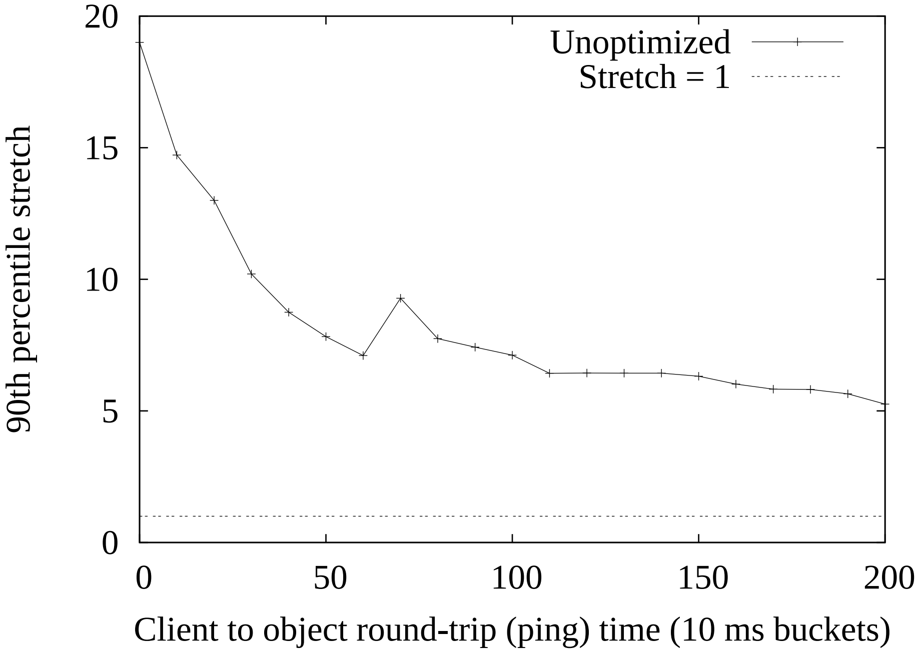

Figure 2.4: *Stretch is higher at short distances than at long distances..*

global, parameter.

In order to get provable bounds, these systems place more object pointers than Tapestry; though proving the necessity of these extra pointers is outside the scope of the thesis, we motivate them in Section 2.4.

The main difference between Pastry (a DHT) and Tapestry (an object location system) is the placement of intermediate pointers. But taking this point of view can be misleading—just placing more pointers in the network does not always reduce stretch. Placing them in the wrong place, for example, won't help. And in some networks, it may be too expensive to place enough pointers to make a significant difference. Understanding these tradeoffs requires studying object location as a problem by itself.

In Figure 2.3, we measure the stretch, and plot it as a function of the distance between the searcher and the object. Notice that stretch is highest when the object and the searcher are nearby, and effect we explain in the next section.

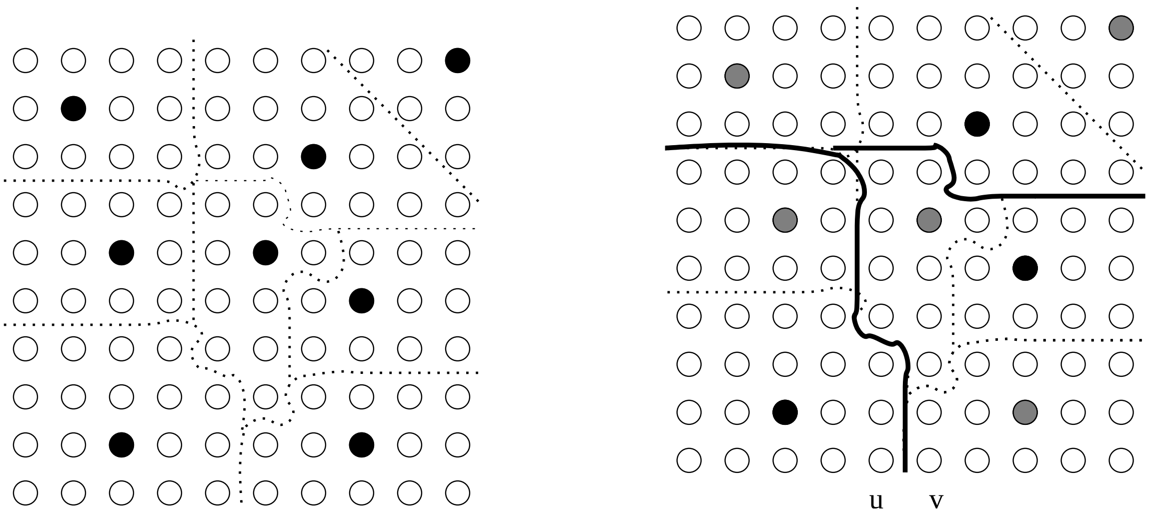

Figure 2.5: *Self-organization into a hierarchy.* The set of nodes with prefix α (here represented in black) partition the network. Two nodes that are close to each other are most often—but not always—in the same region.

2.4 PRR-trees as a hierarchy of directories

The introduction described a generalized hierarchical lookup strategy for object location. Such a scheme checks a local directory first, and then checks the directory for progressively larger regions. Thus, nearby objects are found in local directories. This is the intuition behind many object location systems. In the introduction, we identified three challenges: boundary cases, self organization, and load balance. In this section, we show how PRR-trees tackle these challenges.

Self organization:

The key idea is that every node is a directory at every level of the hierarchy.

Consider, for example, all the nodes of prefix α. Figure 2.5 shows the nodes with prefix α in black, and all the other nodes in white. Say a node v "belongs" to A if the path from v to a node with prefix α ends at A. Every node in the network belongs to some α node, so α nodes are regional directories. The scheme of Section 2.3 used them in exactly that way: if v has a pointer for an object with ID α, v places a pointer to that object on its leader. Likewise, if v is looking for an object with ID α,

v checks the directory at its leader before sending a message outside the region.

Load Balance: The leaders do a limited amount of work because only a small number of object requests go through any given node. That is, the black nodes in the picture only do work for objects with prefix α—for any other prefix, there is a different set of leaders with different partitioning.

Finally, note that $\alpha \circ j$ divide the α nodes rather than directly redividing all the nodes, a node's α leader can determine its $\alpha \circ j$ leader. See Figure 2.5 (right). If this were not so, load balancing the requests would be very expensive, since nodes would have to store all directories for all objects. Under the current scheme, a node need only store the most local directory for every object; if the local directory does not have it, the local directory sends it up the hierarchy. (Since each node plays a role at every level of the hierarchy, it has to know the next level hierarchy for all objects, which amounts to knowing a node with prefix $\alpha \circ j$ for every α that is a prefix of its own ID.) We show a low stretch system for general metrics that does not have this property, and so seems impossible to load balance, in Chapter 6.

Boundary cases: Consider two nodes u and v who route messages through different α nodes (that is, u and v are separated by a dotted line in Figure 2.5). Even though u and v are physically quite close, through bad luck, they do not belong to the same α node. Worse, at the next higher level of the hierarchy (on the right) u and v are still separated, so u takes a long time to find objects published from v and vice-versa, and the stretch is high as a result.

PRR guaranteed low stretch by putting object pointers on the secondary neighbors (at a set of nearby α nodes rather than just the closest) to deal with this. In the interest of reduced complexity and storage, Tapestry gives up on the first problem, thus fails to get constant stretch, though it gets low stretch for most pairs.

Recent work by Stribling, Hildrum, and Kubiatowicz [SHK03] investigated this effect by measuring the stretch of object location requests between nearby pairs on a simulated transit stub topology. Figure 2.6 considers a simulated transit stub

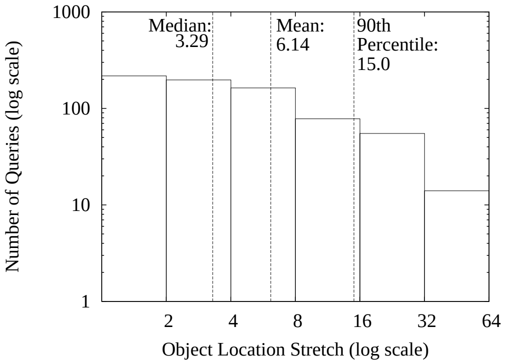

Figure 2.6: *Histogram of stretch for nearby pairs.* This graph shows that while most nearby pairs have low stretch, a significant number have very high stretch.

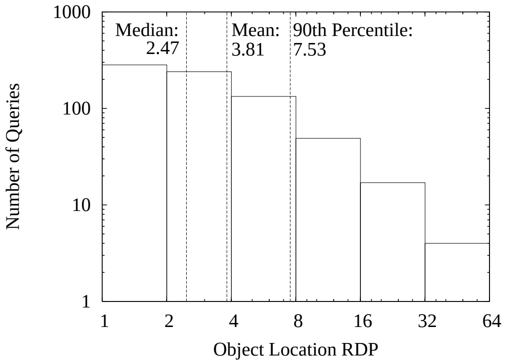

Figure 2.7: *The effect of publishing to backups on stretch.* Note that the stretch has improved significantly as compared to Figure 2.6.

topology and looks only at nearby pairs of nodes. Notice that while most pairs have low stretch, a few have very high stretch. Figure 2.7 shows that publishing to backups (essentially the technique suggested by PRR, though implemented here in a limited way), significantly improves stretch. Note that it is important that both these graphs consider only nearby pairs—if they instead showed all pairs, the stretch would be quite low, since the stretch is trivially low for far away pairs, and most pairs in the network are far away. Appendix A describes a thought experiment that gives intuition for this.

In Figure 2.8, we show the effects of backups, this time as a function of distance between the pairs, and then the effect of putting a few extra pointers on nearest neighbors. Note that these techniques help significantly. Chapter 6 shows that we can do a better job by adapting to the local network properties.

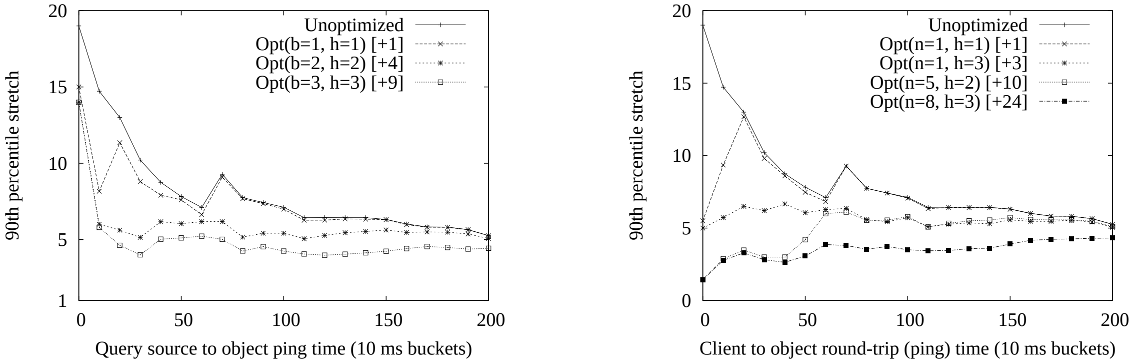

Figure 2.8: *Heuristics to reduce stretch.* On the left, the technique used is publishing to the local backups on a limited basis (b is the number of backups used, h is the number of hops.) On the right, we show the effect of publishing to the nearest neighbors, a sort of limited phone book approach (n is the number of neighbors, h is the number of hops).

Chapter 3

Nodes Joining and Leaving

This chapter gives join and leave algorithms such that the object location data structure described in the previous chapter is maintained as the network evolves. Our approach is repair-oriented, and we give algorithms to return the network to a perfect structure after an event. Section 3.4 outlines the maintenance-oriented technique used by Bamboo.

Recall that the neighbors of a node are divided into neighbor sets, where $\mathcal{N}^A_{\beta,j}$ means the neighbors of A with prefix $\beta \circ j$. We need to ensure that if $\mathcal{N}^A_{\beta,j} = \emptyset$, then there are no nodes with prefix $\beta \circ j$ anywhere in the network (this is Property 1). When this property holds, two messages for ID ψ terminate at the same destination. This chapter gives a join algorithm that maintains Property 1, which means ensuring that a neighbor table entry is empty only when there are no nodes that could fill it. Ensuring that each set contains nearby entries (i.e. maintaining Property 2) is dealt with in Chapter 4.

After a node is finished joining, the network should be the same as if the node had been there when it entered. This means in addition to correctly updating the neighbor tables, maintaining the following invariant:

Property 3. *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.*

Section 3.1 gives an algorithm such that if Property 1 and Property 3 hold, then

```
method Join (gateway, NewNode)
  PSurrogate ← AcquirePrimarySurrogate (gateway, NewNode)
  $\alpha$ ← GreatestCommonPrefix(NewNode, PSurrogate)
  CopyNeighborTable [on PSurrogate] ()
  AcknowledgedMulticast [on PSurrogate]
      ($\alpha$, LinkAndXferRoot[NewNode])
  AcquireNeighborTable (NewNode)
end Join
```

Figure 3.1: *Node Join Routine.* The join 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. Note that *NewNode* or *Psurrogate* is actually a pair of values: the name of the node and its IP address.

```
method AcknowledgedMulticast($\alpha$, Function)
  if NotOnlyNodeWithPrefix($\alpha$)
      for $i = 0$ to $b - 1$
          neighbor ← GetMatchingNeighbor($\alpha \circ i$)
          if neighbor exists
              $\mathcal{S}$ ← AcknowledgedMulticast [on neighbor)] ($\alpha \circ i$, Function )
  else
      apply Function
  wait $\mathcal{S}$
  SendAcknowledgement()
end AcknowledgedMulticast
```

Figure 3.2: *Acknowledged Multicast.* It runs function on all nodes with prefix α.

all three hold after the join. During the node join, 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 3.1.3 shows how the algorithm can be extended to eliminate this problem.

In a real network, joins may happen in bursts. If two nodes entering end up with inconsistent state, and then Proposition 1 does not hold, so section 3.2 extends the original join algorithm to ensure that this does not happen.

3.1 Basic Algorithm

Figure 3.1 shows the basic join algorithm. For the moment, we assume that only one node is inserting at a given point in time. In Section 3.2, we drop this assumption. 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.

First, the new node finds the node with the ID closest to its own. This node is called the surrogate. (In fact, the insertion can be run from any node, but to be efficient, should start from a node that matches the new node in as long a prefix as possible. What follows assumes the insertion starts at such a node.) To find this node, the new node routes toward its own ID. The place where the routing stops is its surrogate. Once it finds its surrogate, it copies the surrogate's neighbor table. 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 3.1.1) to do this. As a final step, we build the neighbor tables, as described in Chapter 4. Notice that the surrogate's table, though it contains far away neighbors, is good enough to let the new node be functional. Thus, once the multicast is finished in step 4, the node is fully functional.

To maintain Property 3, *all* nodes on the path from an object's server to the object's root must have a pointer to that object. Note that as long as a pointer to an object ψ exists on at least the root of ψ ($\mathcal{R}_\psi$), the object is found, though the stretch may be high. Thus, there are two failure cases, one of correctness (Tapestry-as-a-DHT case), in which missing a pointer on $\mathcal{R}_\psi$ means ψ is unavailable, and one of performance, where missing a pointer to ψ on some other node may increase stretch.

The function LinkAndXferRoot from Figure 3.1 takes care of correctness by transferring object pointers that should be rooted at the new node and deleting pointers that should no longer be on the current node. If we do not move the object pointers, then objects may become unreachable. Performance optimization involves redistributing pointers and will be discussed in Section 3.1.2.

3.1.1 Acknowledged Multicast

To contact all nodes with a given prefix we introduce an algorithm called *Acknowledged Multicast*, shown in Figure 3.2. 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 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.

This multicast forms a tree. The origin of the message is the root, and everyone that node sends the message to is a child, and the nodes they send the message to are their children, and so on. Because the new node needs to know when the algorithm is finished, each recipient sends an acknowledgment to its parent (*i.e.*, the sender) after receiving acknowledgments from its children. If a node does not forward the multicast, it sends the acknowledgment immediately. When the initiating node gets an acknowledgment from each of its children, it knows that all nodes with the given prefix have been contacted.

Theorem 2. *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 $O(dk)$.

A stateless multicast: The multicast algorithm described above requires the nodes in the middle of the multicast to maintain state. A modified version of the algorithm puts the burden on the entering node, rather than on the intermediate node. The change to the protocol is that that any node acknowledging the multicast message acknowledges directly to the node that started the multicast and include in the acknowledgment all nodes to which the message was forwarded. Given that information, the initiating node can determine when the multicast has finished, and when and where a message might have been lost. Because acknowledgments from nodes can arrive out of order, it is possible to receive an acknowledgment before knowing that one is expected.

3.1.2 Object Pointers

Recall that publishers put objects 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

```
method OptimizeObjectPtrs (sender, changedNode, objPtr, level)
  $oldsender \leftarrow$ GetOldSender(objPtr)
  if $oldsender \neq null$ and $oldsender \neq sender$
     OptimizeObjectPtrs [on NextHop(objPtr, level)]
                         (self , changedNode, objPtr, $level + 1$)
     if $oldsender \neq changedNode$
        DeletePointersBackward [on oldsender]
                         (objPtr, changedNode, $level - 1$)
end OptimizeObjectPtrs

method DeletePointersBackward (changedNode, objPtr, level)
  $oldsender \leftarrow$ GetOldSender(objPtr)
  delete(objPtr)
  if $oldsender \neq changedNode$
     DeletePointersBackward [on oldsender] (objPtr, changedNode, $level - 1$)
end DeletePointersBackward
```

Figure 3.3: *OptimizeObjectPtrs and its helper function.*

that maintains Property 3. This function is used to rearrange the object pointers any time the routing mesh changes the 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 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 3.3 shows the two methods that are needed to implement this procedure.

Notice, however, that Property 3 is not critical to the functioning of the system.

```
method ObjectNotFound (objectID)
if (Joining)
    level ← length(GreatestCommonPrefix(NewNode, PSurrogate))
    FindObject [on PSurrogate] (objectID, level)
elseif not RoutingConsistentWithNeighbors(objectID)
    RetryRouting(objectID,Neighbors)
endif
end ObjectNotFound
```

Figure 3.4: *Misrouting and route correction.* Misrouting and route correction are used to keep objects available even during the join.

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.

3.1.3 Keeping Objects Available

While a node is joining the network, object requests that go to the new node after the join may either go to the new node or to a pre-join destination. Figure 3.4 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 a joining 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-join root receives a request for an object pointer that has already been moved to the new node, it should forward the request to the new node. But we want to do this in such a way that the surrogate does not need to keep state to show which nodes are joining. So we require all nodes to "check the routing" of an object request or publish before rejecting it: the nodes test whether the object made a surrogate step that it did not need to make. If it finds out it did make a surrogate step instead of going to the new node, the old root node redirects the message to the new node.

To make this work properly, we require that the old root not delete pointers until the new root has acknowledged receiving them. If this is done, then one of the two nodes is guaranteed to have the pointer. No matter which node receives the request, before or after the transfer of pointers, the node servicing the request either has the information to satisfy the query or else it forwards the query to the other node, which can satisfy it using local information.

Finally, it is possible for a request for a non-existent object to loop until the join 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.

A Two Phase Protocol: As an alternative, insertion can be done in two phases. The first phase moves the object pointers, and only in the second phase do the tables get updated. This ensures that the new node has the object pointers before any routing occurs.

3.2 Simultaneous Joins

As mentioned in the introduction to this section, in a wide-area network, joins may not happen one at a time. If two nodes join at the same time, each may get an older view of the network, so neither node will see the other. Suppose A and B join simultaneously. There are three possibilities:

- A's and B's joins 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.

```
method AcknowledgedMulticast
        ($\alpha$, Function, holebeingfilled, watchlist, NewNode)
  watchlist ← CheckForNodesAndSend(watchlist, NewNode)
  if NotOnlyNodeWithPrefix($\alpha$)
     for $i = 0$ to $b - 1$
        neighbor ← GetMatchingNeighbor($\alpha \circ i$)
        if neighbor exists
           $\mathcal{S}$ ← AcknowledgedMulticast [on neighbor] ( $\alpha \circ i$,
                    Function, holebeingfilled, watchlist,NewNode)
  else
     apply Function
     $\mathcal{S}$ ← MulticastToFilledHole(holebeingfilled, Function, watchlist,
                    NewNode)
  wait $\mathcal{S}$
  SendAcknowledgement()
end AcknowledgedMulticast
```

Figure 3.5: *Acknowledged multicast with the watch list.* This version of acknowledged multicast handles simultaneous joins.

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 having having nodes periodically rejoin the network, but before the occurs, objects may be unavailable. This is a serious problem, and this section presents our solution. (Recent work by Liu and Lam [LL03] also addresses this problem; their solution has the advantage that it does not requires that non-joining nodes maintain 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 the join, must start at a core node.

Two operations are *simultaneous* if there is a point in time when both operations are ongoing. This straight forward definition is important from a systems standpoint. It is a bit imprecise, however, since two multicasts could be simultaneous and yet be indistinguishable from sequential multicasts without the use of a global clock. We would like to distinguish between cases where all core nodes see evidence consistent with a sequential ordering and cases where there can be no such agreement. To this end, we say that two multicasts *conflict* if there are two nodes that receive the multicasts in different orders. 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 3. *Suppose A joins. 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 joins 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 1 states that simultaneously joining nodes cannot interfere with one another's access to core nodes.

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

Proof. Proof by contradiction. Theorem 2 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 a (α, j) node in its table. But this is not possible:

Case 1: B has not yet finished its multicast. Since B is supposed to send the multicast to X, we know that B and X 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. But this means that B would only have filled (α, j) entries, so it could not possibly have been contacted with a prefix smaller than $\alpha \circ j$. Contradiction.

Case 2: B has finished its multicast and is a core node. By Theorem 3, since X is an (α, j) node, B must have such a node in its table. Contradiction.

Although Theorem 3 uses Lemma 1, Case 2 is not circular: Node B joined *before* the point in time that we use it here. □

It remains to deal with the case where two joins 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 2. *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 3. *Suppose A and B fill the same hole. Then with the modification described above, if A's multicast conflicts with B's, A will get B's multicast message before B's multicast is finished.*

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

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. When the joining node sends this to the surrogate, it is a zero for every entry in the neighbor table. Each receiving node checks the watch list to see if it can fill in any blank on the list. If it can, it sends the relevant node to the originator of the multicast, marks the entry as found, and continues the multicast. From this description, it may sound as if we are sending a lot of information; in fact, we will be sending very little, since most of the lower levels of the table will be filled by the surrogate in the first step, and most of the upper levels of the table will be zero. In the normal case, we send only a few levels of the neighbor table, and each level is sixteen bits. This new version is shown in Figure 3.5. Using this new multicast, we get Lemma 4.

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

Proof. There are two cases:

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

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

This completes the proof. □

Finally, we put everything together and prove Theorem 3.

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

Case 1: If A and B fill different holes on the same level (i.e., A fills an (α, i) hole and B fills (α, j) hole for $i \neq j$) , then they multicast to the same prefix α. By

Lemma 1, we know these nodes are reached, and we can apply Lemma 4, once with A in the theorem being A of the lemma, and once with A 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 1 we know these nodes are reached. Given that, we again apply Lemma 4.

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 3, which says that if the two multicasts are not serialized, each will find out about the other before their multicasts complete.

This completes the proof. □

3.2.1 Discussion

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

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

Liu and Lam [LL03] present a different algorithm for simultaenous joins in a PRR-like network. In their scheme, an inserting node may sometimes have to wait for other nodes to finish insertion before going to the next phase, while in our scheme, no insertion ever waits on the completion of another insertion. One nice feature of their scheme is that their notification algorithm (roughly the equivalent of our multicast algorithm) has the property that only the inserting node must keep state. They get

```
method DeleteSelf ()
    for pointer in { backpointers }
        level = GetLevel(pointer)
        LeavingNetwork [on GetIP(pointer)] (selfID, level, GetNearest(pointer, le
    for pointer in {neighbors $\cup$ backpointers }
        RemoveLink [on GetIP(pointer)] (selfID)
end DeleteSelf
```

Figure 3.6: *Voluntary Delete.* This shows what a node should do when it leaves the network.

this property by having the inserting node copy the neighbor table of every node it contacts.

3.2.2 Running Time Analysis

Finding the surrogate is no more costly than searching for an object pointer, and [PRR97] argues 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. But k will be small in expectation, and bounded by $\log n$ with high probability. 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.

3.3 Departing Nodes

In this section, we present algorithms that help maintain our invariants when nodes leave the network. We consider two cases: *voluntary* and *involuntary* exits. A *voluntary* exit occurs when a node informs the network that it is about to exit. This is the preferred mode of exit that permits the infrastructure to maintain availability of objects by fixing neighbor links and object pointers. An *involuntary* exit occurs when a node ceases to 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 exits are voluntary departures, and we present an algorithm for this case only for completeness. In real networks, nodes and links will typically fail without warning, so involuntary exit is the common case. We discuss this case in Section 3.3.2.

3.3.1 Voluntary Departure

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 3.6.

After node A sends out its initial notification messages, it examines local object pointers for which it is the root, and forwards them on 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 exit 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.

3.3.2 Involuntary Departure

Involuntary departures occur 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 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.

Notice that we cannot proactively respond to involuntary exits because there is no notification. Some link just stops working. Whatever repair technique is used, there must be some redundancy built into the network to tolerate faults, or the network breaks any time someone pulls the power cord. The key challenge is to determine the right level of redundancy to ensure that the network work continues to work until the repair routine can complete.

We propose that unexpected exits 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. Exits can be detected by soft-state beacons [ZHR$^+$03] 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 departed node.

To ensure Property 1, if exiting 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 [LL03] 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 that only the starting node to maintain 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.

3.4 Maintenance vs Repair

As we mentioned in earlier, there are two approaches to changes in peer-to-peer networks. This chapter took the repair approach, giving algorithms for node joins and leaves. This section gives a flavor of the other approach by focusing on Bamboo [RGRK, RGRK03]. Another way to describe this is to say that to improve availablity, the system desinger can either increase mean time between failure (MTBF), or decrease mean time to repair (MTTR), or both. The proactive approach focuses on the latter, and the maintance approach focuses on the former.

In particular, the maintenance approaches add redundancy to the network so that it functions even when slightly imperfect. As a result, proactively fixing problems (like holes in the neighbor table) becomes less important. Allowing for simple periodic maintenance mechanisms that return the network to the correct state in a reasonable period of time. In highly dynamic environment, this approach seems to make more sense. Liben-Nowell, Balakrishnan, and Karger presented maintenance algorithms for Chord in [LNBK02a, LNBK02b]. Rather than building a neighbor table for a node in Chord, they gave procedures to be run at a fixed interval. Over enough time, these procedures stabilize the network. These are the procedures in the most recent presentations of Chord [SMLN$^+$03]. Bamboo [RGRK, RGRK03] takes a similar approach, which we describe in detail.

Maintenance in Bamboo: Bamboo [RGRK, RGRK03] has the same structure as PRR, Tapestry, and Pastry. Bamboo has the same neighbor tables as PRR with the addition of a Pastry-style leaf set, keeping the nodes with the closest k IDS in both directions. And object is stored not only on the node most closely matching its ID, but on the leaf set of that node. The result is same structure as Pastry.

The key idea used by Bamboo (and by Chord [LNBK02b, LNBK02a, SMLN$^+$03]) is to separate the necessary from the optimizations, and make the necessary as small and simple to maintain as possible. For Bamboo, the necessary is exactly the leaf set (for Chord, it is the successor list).

To see this, notice that given only the leaf set, Bamboo still functions, if only inefficiently. As we noted earlier, the key operation for DHTs is to route to a given key, and Bamboo can do this with only the leaf set. If the destination ID is above the current ID, use a node from the top half of the leaf set, if it is below the current ID, use a node from the bottom half of the leaf set. As long as at least one node in either direction exists, every routing step makes forward progress.

To maintain the leaf sets, nodes in Bamboo periodically contact the members of their leaf set to compare leaf sets and exchange object pointers. If a node discovers during this phase that there is node that should be in its leaf set, or an object pointer it should be hold, it adds it. This extremely simple technique handles node join and node departure smoothly.

In particular,

- A node enters, routes toward its ID, takes the leaf set and neighbor table, and is then fully functional. This requires only one round trip message! As this node starts contacting members of its leaf set via the maintenance process, other members of its leaf set find out about the new node.
- Suppose a node disappears. When members of its leaf set attempt to contact the now deceased node, they will not get through, but through normal communication with their leaf set, they replace the node.
- Object pointers maintenance is efficient. If a node's object pointers all disappeared, they would be slowly replaced via communication with the leaf set.

This network is not invulnerable, but the entire leaf set in one direction must fail before the simple repair mechanism mentioned above cannot fix it.

Concentrating on the leaf set as the correctness piece allow for simplicity in other areas. In the Tapestry neighbor table, having a hole is a problem, since there is no alternate routing path. (It also affects surrogate routing, but this can be overcome by using a root set as described above.) Tapestry's neighbor tables used R (set to three for Tapestry) per slot to make holes rare. But in Bamboo, the whole neighbor table is merely an optimization, so Bamboo need only keep one node per entry, and uses the leaf set otherwise.

Finally, this way of dealing with dynamism does not seem to affect performance. In addition, the routing latency for Bamboo is also low, see [RGRK, RGRK03]. KRISLOOKTHISUP.

The repair approach (used by Pastry [RD01a], CAN [RFH$^+$01], and the original Chord [SMK$^+$01] as well as here) is probably the better approach when changes are rare and lots of bandwidth is available. The maintenance-oriented approach [LNBK02b, LNBK02a, SMLN$^+$03, RGRK, RGRK03] is probably better when changes are frequent or bandwidth is limited.

Chapter 4

A Nearest Neighbor Algorithm for Growth-Restricted Metrics

As we explain in the next few paragraphs, building the neighbor tables for a given node requires a subroutine to find the nearest node in the network. This chapter shows an algorithm for find the nearest neighbor in a growth-restricted (*i.e.*, grid-like for some dimension d) network.

Recall that nodes have been assigned random IDs, which we think of has being in base b, and building the neighbor tables for a new node A requires determining the the closest $\beta \circ j$ nodes to A for every βs that is a prefix of A's ID. In this chapter, we show how to find the closest R such nodes with prefix $\beta \circ j$, thus maintaining locality property (Property 2). This is necessary to for low-stretch object location.

An efficient algorithm to do this also gives an efficient algorithm to find A's nearest neighbor in the network. (Since the closest among $\cup_j \mathcal{N}^A_{\epsilon,j}$ is A's nearest neighbor.)

This chapter gives algorithms that can be applied to either PRR [PRR97] or Tapestry. Modifications of these algorithms were used in [AMD04]. These algorithms were implemented as part of Tapestry. The simple algorithm of Section 4.1 appeared in [HKRZ02], and the improved algorithm in Section 4.2 appeared in [HKR03, HKMR04]. The first algorithms requires no more space than that already present in the Tapestry routing data structure, though the second requires an additive $O(\log n)$ storage in the dynamic case. The main idea is to use the routing structure backward.

```
method AcquireNeighborTable (NewNode,PSurrogate)
  $\alpha \leftarrow$ GreatestCommonPrefix(NewNode, PSurrogate)
  maxLevel $\leftarrow$ Length($\alpha$)
  list $\leftarrow$ AcknowledgedMulticast [on PSurrogate] ($\alpha$,
        SendID(NewNode, NewNode))
  BuildTableFromList(list, maxLevel)
  for  $i$ = maxlevel - 1 to  0
       list $\leftarrow$ GetNextList(list, $i$, NewNodeName, NewNodeIP)
       BuildTableFromList(list, $i$)
end AcquireNeighborTable

method GetNextList (neighborlist, level, NewNodeName, NewNodeIP)
  nextList $\leftarrow \emptyset$
  for $n \in$ neighborlist
      temp $\leftarrow$ GetForwardAndBackPointers($n$, level))
      AddToTableIfCloser [on $n$] (NewNodeName, NewNodeIP)
      nextList $\leftarrow$ KeepClosestk(temp $\cup$ nextList)
  return  nextList
end GetNextList
```

Figure 4.1: *Building a Neighbor Table.* The AcknowledgedMulticast function is described in Figure 3.2.

Other PRR-based routing systems use heuristic techniques to build their neighbor tables. Pastry [RD01a] and the original Tapestry [ZHR$^+$03] use a physically nearby node to get a good starting neighbor table. Then, they optimize the table using local search. In particular, a node asks its neighbors for their neighbors, and then measures the distance to those nodes, and picks the closest for its new neighbor set. Another technique, described by Gummadi et al. [GGG$^+$03] is to pick k random nodes and use the closest. That is, for $\mathcal{N}^A_{\beta,j}$, pick k random IDs with prefix $\beta \circ j$, route to those IDs, and measure the distance from those nodes to A, keeping the closest. Bamboo [RGRK, RGRK03] compares these techniques with a local search algorithm inspired by the algorithm presented in a highly dynamic network.

One provable solution is to consider the subnets formed by each prefix α, and run any nearest-neighbor search algorithm on those subnetworks. Each node only participates in $O(\log n)$ subnetworks, resulting in a blow up of $O(\log n)$ compared to the generic neighbor search algorithm.

While finding the nearest neighbor is a well-studied problem, there are some special properties here. The points do not have coordinates in Euclidean space, so techniques that use coordinates (e.g., *kd*-trees [Ben75]).

We are limited to oracle or black-box methods, in which the algorithm is allowed to ask for the distance between two points, and that is the only thing it is allowed to do. In a real network, we implement the oracle by pinging a node. Also, the data structure must be distributed and load-balanced. A naive implementation of *vp*-trees [Yia93] places a lot of load on one particular peer that happens to be the root. In some sense, the algorithm presented here i produces load-balanced vp-trees.

Finally, finding the nearest neighbor is a hard problem in a general network, and could require contacting every node. Consider, for example, a network where all peers are at distance one from each other. A new node enters, and it is at distance one from every node except one. Finding that node requires $\Omega(n)$ queries to the distance oracle.

Recall, however, that PRR's object location solution applies in a limited class of metric spaces. This class admits efficient algorithms for nearest neighbor search. We actually consider a slightly larger class of metric spaces, called *growth-restricted.* This is formally defined in 4.1. Karger and Ruhl [KR02] gave the first algorithm for nearest neighbor search in growth restricted spaces. Their nearest neighbor search structure requires $O(\log n)$ space and runs in $O(\log n)$ time. The overall idea is to halve the distance between the current point and the query point. After $\log n$ successful halving steps, the algorithm finds the nearest neighbor of the query point. To implement these halving steps, they use a random permutation of the nodes, and a data structure that related to Chord [SMLN$^+$03]. The inspiration for the algorithms described in this chapter follows this very general outline, but the resulting algorithms are different.

This chapter gives two similar algorithms. Section 4.1 gives a conceptually simple algorithm that takes $O(\log^2 n)$ messages. Section 4.2 uses a similar algorithm and a more careful analysis to get $O(\log n)$ messages. A bonus of these algorithms is that finding the nearest neighbor for a given node finds *all* the entries in the neighbor table. In contrast, using the techniques of Karger and Ruhl requires data structures for each ID prefix, requiring $O(\log^2 n)$ space in total.

In the non-distributed case, both of our algorithms use only linear space. In this setting, the data structure is similar to the deterministic algorithm of Krauthgamer and Lee [KL04b]. Their algorithm has applications in a broader class of metric spaces, though it is not yet clear whether it can be used as a distributed algorithm.

The overall approach is similar to that of Clarkson in [Cla97], and the sampling technique used by Thorup and Zwick [TZ01] for approximate distance oracles is similar to our technique. We also note that the general idea of our algorithm is very similar to the idea used by Plaxton, Rajaraman and Richa [PRR97] to find a nearby copy of an object.

As in [PRR97, KR02], we adopt the following network constraint. Let $\mathcal{B}_A(r)$ denote the ball of radius r around A; *i.e.*, all points within distance r of A, and $|\mathcal{B}_A(r)|$ denote the number of such points. We assume:

$$|\mathcal{B}_A(2r)| \leq c\,|\mathcal{B}_A(r)|\,, \tag{4.1}$$

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 [KR02]. 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 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.

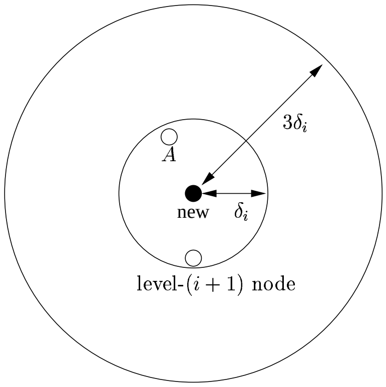

Figure 4.2: *Circle Lemma 4.* If $3\delta_i$ is less than δ_{i+1}, then A must point to a node within δ_{i+1}.

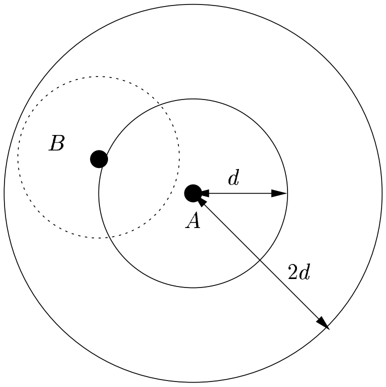

Figure 4.3: *Figure for Theorem 5.* The larger ball around A contains $O(\log n)$ nodes, while the smaller ball contains none.

4.1 A simple scheme

The idea is fairly simple. Figure 4.1 shows how to build neighbor tables. Suppose that the longest common prefix of the new node and any other node in the network is α. Then we begin with the list of all nodes with prefix α. We get this list using AcknowledgedMulticast, described in Chapter 3. Then we get the lists for progressively smaller prefixes, until we have the closest k nodes matching the empty prefix. The level-i list fills in level-i of the neighbor table.

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 us all the level-i nodes they know of (we ask for both forward and backward 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 joining node. Each node contacted this way also checks to see if the new node should be added to its own table (line 4). We then trim this list, keeping only the closest k nodes. If $b > c^2$, then Lemma 5 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 line 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 is the first i digits of the node's prefix. To fill in level i of the neighbor table, we look in the level-i list. For $j \in [0, b-1]$, we keep the closest R (α_i, j) nodes (R is defined in Section 2.1), and call the set $\mathcal{N}^A_{\beta,j}$. The algorithm presented here is here is sensitive to failures, a slight modification can make the algorithm substantially more robust, see Chapter 5.

4.1.1 A Proof of Correctness

Theorems 4 and 5 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 4 uses Lemmas 5 and 6 to show that the new node's table gets built correctly, and Theorem 5 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 5. *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 $k \geq \frac{24(a+1)b\log n}{(1-c^2/b)^2}$, 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 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 an 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 4.2.) 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 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 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:

$$\begin{aligned} \lambda = & \ \tfrac{1}{2}(1 - c^2/b) < \tfrac{1}{2} \\ \lambda' = & \ \lambda(2 - c^2/b) < 1 \end{aligned}$$

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 $k = \frac{6(a+1)b\log n}{\lambda^2}$.

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 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

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

We 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.

But

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

Using a Chernoff bound, this is less than

$$\exp(-\lambda'^2 \mathrm{E}[X_l]/2) \leq \exp(-\lambda'^2 k/2) \leq \exp(-\lambda^2 k/2).$$

Substituting for k, this becomes

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

We show $\Pr[3\delta_i \geq \delta_{i+1} \,|\, L_{\text{real}} \leq l] \leq 2/n^{a+1}$. 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 Equation 4.1 and the fact $L_{\text{real}} \leq l$. Let Y_m be the number of $i+1$ nodes in m trials. Then, rewriting our goal with this notation, we wish to bound $\Pr[Y_{c^2 l} \geq k \,|\, X_l \geq k]$. (Notice that $Y_{c^2 l}$ is not independent of X_l.) We can write that the $\Pr[A \,|\, B] = \frac{\Pr[A \cap B]}{\Pr[B]} \leq \frac{\Pr[A]}{\Pr[B]}$, so it suffices to bound $\frac{\Pr[Y_{c^2 l} \geq k]}{\Pr[X_l \geq k]}$. We already bounded the denominator, so next we wish to bound $\Pr[Y_{c^2 l} \geq k]$.

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

$$\Pr[Y_{c^2 l} \geq k] = \Pr[Y_{c^2 l} \geq (1+\lambda)\mathrm{E}[Y_{c^2 l}]] \leq \exp(-\lambda^2 \mathrm{E}[Y_{c^2 l}]/3).$$

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

So $\Pr[3\delta_i \geq \delta_{i+1} \,|L_{\text{real}} \leq l] \leq \frac{\Pr[Y_{c^2l} \geq k]}{\Pr[X_l \geq k]} \leq \frac{1/n^{a+1}}{1-1/n^{a+1}} \leq \frac{2}{n^{a+1}}$. (So long as $n^{a+1} > 2$)

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

$$\begin{aligned}
&\Pr[3\delta_i \geq \delta_{i+1}] = \\
&\qquad \Pr[3\delta_i \geq \delta_{i+1} \,|L_{\text{real}} > l]\Pr[L_{\text{real}} > l] \\
&\qquad + \Pr[3\delta_i \geq \delta_{i+1} \,|L_{\text{real}} \leq l]\Pr[L_{\text{real}} \leq l] \\
&\leq \quad 1 \cdot 1/n^{a+1} + 2/n^{a+1} \cdot 1 \\
&\leq \quad 1/n^a
\end{aligned}$$

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

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 6. *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 $k \geq 8(a+1)b\log n$, 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 $\Pr[X_j \leq R]$, and since $R = o(\log n)$, we have $R \leq \frac{1}{2}k/b$.

Thus,

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

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

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 4. *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 4.1 will produce the new node's correct neighbor table with probability $1/n^a$ for any constant a.*

Proof. By Lemma 5, 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 $\frac{\log n}{2n^{a+1}} \leq 1/(2n^a)$.

Second, by Lemma 6, 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 $\frac{\log n}{2n^{a+1}} \leq 1/2n^a$.

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 Figure 4.1. It remains to show that with high probability, line 4 of Figure 4.1 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 5. *If a new node B is a (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 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 4.3). 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 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 probability that $|\mathcal{B}_A(d)|_{(\alpha,j)} \leq R$ and $|\mathcal{B}_A(2d)|_\alpha \geq k$ 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 a (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 $X_{l_{\text{real}}} \leq R$. Then we can say that $\Pr[X_{l_{\text{real}}} \leq R] \leq \Pr[X_l \leq R]$. Further $\mathrm{E}[X_l] = p(8a/p)\log n = 8a\log n$, so $\Pr[X_l \leq R] \leq \Pr[X_l \leq (1-\frac{1}{2})\mathrm{E}[X_l]]$. Using a Chernoff bound, this is less than $\exp(-\frac{1}{2}^2\mathrm{E}[X_l]/2)$, and substituting, this is less than $\exp(-\frac{1}{2}^2 8a\log n/2) \leq \exp(-a\log n) = 1/n^a$.

Case 2: $l_{\text{real}} \leq l$. In this case, we argue that the ball of radius $2d$ around A contains more than k nodes with probability less than $1/n^a$. But 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 a random variable representing the number of α nodes in m trials. We wish to bound

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

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

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

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

$$\Pr[Y_{cl} \geq 2\mathrm{E}[Y_{cl}]] \leq \exp(-\frac{8}{3}abc\log n) \leq \exp(-a\log n).$$

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 4), 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 5).

Finally, choose $k = \max(k_1, k_2)$, and the probability that the algorithm of Figure 4.1 fails to perform the correct updates will be less than $1/(2n^a) + 1/(2n^a) \leq 1/n^a$.

4.1.2 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. But this analysis can be tightened. Using the techniques of Theorems 4 and Theorem 5, 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 $O(qd) = O(d \log n)$.

4.2 A better scheme

This section gives a similar algorithm that allows for a tighter analysis. This algorithm only contacts $O(\log n)$ nodes, rather than $O(\log^2 n)$. To simplify the presentation, this section considers only the problem of finding the nearest neighbor, and ignore the issues of filling in the neighbor table.

The idea of the algorithm is as before. That is, it starts with the root node in the current list. Then it asks each node in the current list for their children. Choose the children close enough to have descendants that are "close" to be the in the next level current list. Then query those, and so on. The problem is determining which nodes could have nearby descendants. The version in Section 4.1 $O(\log n)$ at each step, but this is overkill. In expectation, only a constant number need to be kept. Keeping just a constant number does not work, however, because some level needs $O(\log n)$, so this section analyzes all levels at once.

4.2.1 Using Hints

Suppose the algorithm gets hints. In particular, assume the algorithm knows the distance to the closest node at each level. We analysis the performance of algorithm with these hints, and then we show why the hints aren't needed.

For an index i, let $d_i(x)$ be the distance from x to the closest level i node. We drop the x when clear from context.

Let $q_0(x) = d_0(x)$, and for $i > 0$, let $q_i(x) = \max(3d_i(x), 3q_{i-1}(x))$. Lemma 7 shows that all level-$(i-1)$ nodes within $q_{i-1}(x)$ have parents within $q_i(x)$ of the query node. The proof is very similar to Lemma 5.

Given the q_is and the single $\log_b n$ level root node, we can find the nearest neighbor of a node x, as follows. First, x queries the root for its children, and keep all the children within q_i for the i corresponding to that level. For all those nodes, query their children, and keep the children within q_{i-1} and so on. Pseudocode for this case is shown in Figure 4.5. In Section 4.3, we show how the q_i's can be found.

Lemma 7. *If we query all the level i nodes within q_i of x, then we can find all the level $i-1$ nodes within q_{i-1} of x.*

Proof:

We ask all the level i nodes within q_i for their children (the level $i-1$ nodes that point to them). We want to prove that no level-$(i-1)$ node within distance q_{i-1} is

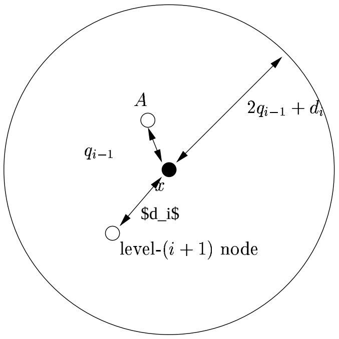

Figure 4.4: *The circle lemma.* The parent of A must lie within the big circle.

```
method FindNearestNeighbor ($rootNode$, $\mathbf{q}$)
  currList ← [$rootNode$]
  nextList ← $\emptyset$
  for $i = maxLevel$ to 1
      for $n \in$ currList
          nextList ← nextList $\cup$ GetChildren($n$)
      currList ← KeepWithDist(nextList, $q_i$)
      nextList ← $\emptyset$
  return (currList)
end FindNearestNeighbor
```

Figure 4.5: *Finding the nearest neighbor with hints.* If the q_i's are given, finding the nearest neighbor is straightforward.

missed.

Suppose A is a level $i - 1$ node within distance q_{i-1} of x. When the parent of A is within q_i of x, then we query A's parent and so find A. Next, we show that $d(A, \text{parent}(A)) < q_i$.

Figure 4.4 shows this situation. Notice that A's parent must be closer to A than the closest level i node to the query point (call this node u_i), since A chooses the closest among the level i nodes. Mathematically, we know that $d(A, \text{parent}(A)) \leq d_i + q_{i-1}$.

We know that the distance between A and the query point is bounded by q_{i-1}, so the distance between the query point and A's parent is bounded by $2q_{i-1} + d_i$, but q_i was chosen to be greater than $2q_{i-1} + d_i$, so we are done. □

4.2.2 Bounding the number of nodes

In this section, we bound the number of nodes contacted during this process. To facilitate this, we define the notion of a certificate. A certificate for x has for each i, a list of all the level-i nodes within distance $q_i(x)$. We view the certificate as being divided up in pieces, called subcertificates. Two adjacent levels i and $i-1$ are in the same subcertificate if $q_i = 3q_{i-1}$. The lowest level in a subcertificate is called a base level. By definition, level 0 is always a base level.

The certificate described above has $O(\log n)$ nodes in expectation if the metric space is growth restricted. To show this, we start with the following Lemma.

Lemma 8. *Suppose i is a base level (i.e., the lowest level in some subcertificate), and the base ball has volume s. Then the expected size of that subcertificate is bounded by $O(s/b^i)$, provided that $c^2 < b$.*

Proof: For a given j, we must find all the level $i+j$ nodes within a factor 3^{j+1} times the original radius. If the original ball had volume s, then each factor of 3 increase in radius increases the volume of the ball by no more than a factor of c^2. So the ball of radius $3^{j+1}d_i$ has volume bounded by $s_i(c^2)^{j+1}$, where d_i and s_i are the base radius and base volume, respectively. For a given j, we only need to store the level $(i+j)$ nodes. The probability that a node is an $i+j$ node is $b^{-(i+j)}$. Combining these two facts with a little algebra, we expect to have no more than $s/b^{i-1}(c^2/b)^{j+1}$ level $(j+i)$ nodes in the certificate. Summing over all possible j, this gives an upper bound of $O(s/b^{i-1})$ nodes in the certificate for base level i, and since b is a constant, $O(s/b^{i-1}) = O(s/b^i)$. (This final step makes a later proof a bit tidier.) □

Now, we can prove the main size lemma.

Lemma 9. *The total expected size of the certificate is $O(\log n)$ if b is larger than c^2, where c is the expansion constant of the network.*

Proof. We bound the total size of a subcertificate at level i by considering the expected size of the subcertificate when there is no base level larger than i. This is an overcount since some levels may be charged to more than one base level. (In particular, every level is charged to base level 0.)

Let s_i be the size of the base ball at level i. If $s_i = s$, that means the first s nodes were not part of the ith sample. Using this fact, we get

$$\Pr[s_i > cb^i] \leq (1 - 1/b^i)^{cb^i} \leq e^{-c}.$$

Now, we know that

$$\begin{aligned}
\mathrm{E}[\text{baselevel}_i] &= \sum_{k=1}^{\infty} \mathrm{E}[\text{baselevel}_i \,\big|(k-1)b_i \leq s_i < kb^i]\Pr[(k-1) * b^i \leq s_i < k \cdot b^i] \\
&\leq \sum e^{-k} kb^i / b^{i-1}.
\end{aligned}$$

For fixed b, this is a constant.

Finally, since there are at most $\log n$ subcertificates, the total certificate size is $O(\log n)$ □

4.2.3 High Probability Bound

The previous section bounded the expected certificate size. In this section, we show that the certificate size is $O(\log n)$ with high probability.

Lemma 10. *Suppose we are given the base ball sizes $s_1 \ldots s_k$. Then with high probability, the size of the certificate is less than $O(\sum_i s_i/b^i)$*

Proof: We want to argue that we can view each base level independently. If that is the case, then we can apply a Chernoff bound, as before, as we are done. The problem is that the levels are not independent.

But consider the following related variable X, where $X = \sum_{i,j} X_j^{(i)}$ and $X_j^{(i)}$ is one if node j is in a subcertificate for level i with base ball s_i. Then X is the sum of independent random variables, and we can apply the Chernoff bound. Since $\mathrm{E}[X] = \sum s_i/b_i = O(\log n)$, we get a high probability bound.

Now, consider the random variable $Y = \sum Y_i$, where Y_i is one if node i is in the certificate. Notice that $\Pr[Y \geq k] \leq \Pr[X \geq k]$, and we bound X by the use of a Chernoff bound. □

Next, we bound the probability that $\sum_i s_i/b^{i-1}$ is large.

Lemma 11. *For the s_i defined as before, $\sum_i s_i/b^{i-1}$ is $O(\log n)$.*

Proof Let $S = \sum_i s_i/b^i$. The probability of a given configuration $s_1, s_2, \ldots s_k$ is

$$\begin{aligned}
(1-1/b)^{s_1}\prod_i (1-1/b^i)^{s_i-s_{i-1}} &\leq \exp\left(-s_1/b_1 + -\sum_{i=2}^{k}(s_i - s_{i-1})/b^i\right) \\
&= \exp(-S + 1/b(S - s_k)) \\
&\leq \exp(-S + S/b) \\
&\leq \exp(-S(1-1/b)) \\
&\leq \exp(-\frac{1}{2}S)
\end{aligned}$$

The number of ways to get a given sum S from k terms is S choose k, so the probability that of a sum is S is less than $(Se/k)^k \exp(-\frac{1}{2}S)$.

Now, let S_0 be such that $S_0/k > 4\log(S_0 e/k)$. Then

$$\begin{aligned}
&(S_0 e/k)^k \exp(-\frac{1}{2}S_0) \\
&\quad = \exp(-\frac{1}{2}S_0 + k\log(S_0 e/k)) \\
&\quad \leq \exp(-\frac{1}{2}S_0 + k\frac{1}{4}S_0/k) \\
&\quad \leq \exp(-\frac{1}{4}S_0)
\end{aligned}$$

Finally, note that the $\Pr[S \geq S_O] \leq \sum_{S \geq S_0} \exp(-\frac{1}{4}S) \leq \exp(-\frac{1}{4}S_0)\sum_i (e^{\frac{1}{4}})^i \leq 5\exp(-\frac{1}{4}S_0)$. This means that when $S = O(\log n)$, the certificate size is greater than S with probability one over polynomial in n.

4.3 Finding the Hints

This section explains how to find the nearest neighbor without knowing the q_i's in advance. The key idea is to use an algorithm that contacts only nodes that are children of those in the certificate. Since each node has an expected constant number

```
method GetClosest ($x$,rootNode, maxLevel)
  for $i = 0$ to maxLevel - 1
      InitializeHeap(heap[i])
      closest[i] $\leftarrow \infty$
  Insert(heap[maxLevel], rootNode, $d(x$,rootNode))
return GetNext($x$, 0, $\infty$)
end GetCertificate

method GetNext ($x$, level, maxDist)
  if AllHigherLevelsEmpty(heap[i]) then return null
  do
      if peek(heap[level]) $<$ maxDist then
          nextDist $\leftarrow$ 3$\cdot$Max(closest[level + 1], Peek(heap[level]))
      else
          nextDist $\leftarrow$ 3$\cdot$Max(closest[level + 1], maxDist)
      next $\leftarrow$ getNext($x$,level + 1, nextDist)
      if notNull(next) then
          if closest[(level + 1)] $= \infty$ then closest[(level + 1)] $\leftarrow$ d($x$, next)
          AddNodeToCertAndChildrenToHeap(next,level)
  while (notNull(next))
  if peek(heap[level]) $<$ maxDist
    return getMin(heap[level])
  else
    return null
end GetNext
```

Figure 4.6: *Finding the certificate and no more.* Anything that leaves a heap is part of the certificate.

of children, if the certificate size is $O(\log n)$, in expectation, we do not contact more than $O(b \log n)$ in the certificate generation process.

To show that the certificate size is bounded with high probability, note that if A is a level-i node within distance q_i, only its children within distance $2q_i$ could matter, since any child at distance $2q_i$ from A is at least q_i from x. What this means is that no children further than $2q_i$ from a level-i in the certificate need to be queried. To get a high probability bound on the number of nodes contacted, we bound the number of level-i nodes within $2q_{i+1}$ of x. This can be done using the same technique as Lemma 10, where the constant in the big-O would be different.

The key idea is that with the ability to access the nodes on a given level in order of their distance from the query point the problem is solved. Given such an ability, starting with $i = 0$, we find the closet node at level i, which we can use to compute q_i.

Figure 4.6 shows pseudo-code for the hintless algorithm. The algorithm maintains, for each level, a list of candidates for the certificate. Nodes removed from these candidate lists are placed in the certificate, and their children become candidates (see line 10). To ensure that this does not place any extra nodes in the certificate, we require that when a level-i node is added to the certificate, there are no closer level-i nodes not already in the certificate. This may require a check for nearby level-$(i + 1)$ nodes, which may require a check for nearby level-$(i + 2)$ nodes, etc. In Figure 4.6, the candidate lists are implemented as heaps, since the algorithm only needs pull out or peek at the minimum.

To guarantee that we have the closest possible, use Lemma 7 as follows. Before moving a level-i node from the candidate list to the certificate, we ensure that there are no unexplored (or candidate) level-$(i + 1)$ nodes that could have a child closer to the query point than the node at the top of the level-i heap. Before arguing this is correct, we point out two facts.

Fact 1. *If a node A is in the the certificate for a query point x, then the parent of A is also in the certificate.*

If we restate this, it says that if a level-i node A is within q_i, the parent of A is within q_{i+1}, which is true by construction and Lemma 7.

Fact 2. *If a level-i node A is in the certificate, then any level-i node B closer to x than A is also in the certificate.*

To see this, recall that the certificate is defined by the q_i's, so if A is in the certificate, $d(A, x) \leq q_i$, but since $d(B, x) \leq d(A, x)$, then $d(B, x) \leq q_i$, so B is also in the certificate.

The algorithm in Figure 4.6 will work by pulling out level-i nodes out of the level-i heap in order of their distance from x. The function is GETCLOSEST does some initialization of global variables, and then calls the recursive GETNEXT. For each level, we keep the distance to the closest found at a given level (or $d_i(x)$). Also, for each level, there's a list of nodes whose parents are in the certificate, but who are not themselves in the certificate yet. We will call these candidate nodes.

The function GETNEXT takes the query point, x, a level l, and a distance $d_{\max}$. Then, if there is a level-l node within the given distance, it will return the closest such node not already in the certificate, and otherwise returns null.

Suppose GETNEXT is called with the query point x, the level is l, and the distance set to d_{max}. Before returning the closest node among the candidate level-l nodes, it must guarantee that there are no closer level-l nodes. By Lemma 7, we get a bound on the distance between x and any the parent of any closer level-l node. Then, we call GETNEXT with the query point x, level $(l+1)$ and this distance bound. If nothing is found, then we know that the candidate is the correct one, and return it. If something is found, we add it to the certificate, add its children to the list of candidates, and check again whether the closest candidate is the closest overall.

To find the nearest neighbor, GETCLOSEST calls GETNEXT with query point x, level 0, and the distance ∞ in line 3. This will cause many recursive calls to GETNEXT, and each such call returns a node in the certificate. By Fact 2, it is enough to argue that the last node returned by GETNEXT on level l is in the certificate, since by construction, GETNEXT returns level l nodes in order of their distance from the query source.

Lemma 12. *The last level-l node returned by* GETNEXT *is in the certificate.*

Proof. The proof is by induction on l. The base case is the nearest level-0 node, which is also the last level-0 node returned, and is clearly in the certificate. Now assume the last level-$(l-1)$ returned is in the certificate. We show that the last level-l node returned must also be in the certificate.

Let A be the last level l node returned by a call to GetNext. There are two cases: The last level-$(l-1)$ is a child of A, or it is not. If it is a child of A, then clearly, A must be in the certificate by fact 1. So suppose it is not. Then we will argue that $d_{\max}$ is no more than q_l, so if A is returned, its distance to x is less than $d_{\max}$ and so less than q_l.

Consider the call stack, and particularly the $d_{\max}$ at level-$(l-1)$. Since we know that the last level-$(l-1)$ node is not a child of A, it must already be in the heap. If it will be returned, by the induction hypothesis, it is at distance less than q_{l-1}, and so that implies that $d_{\max}$ is less than q_l. But suppose it has already been returned. Then we can bound the $d_{\max}$ on the recursive call in a similar way, and repeat the argument backward. □

A simpler, but not as provably good, approach appears in the appendix.

While this describes how to find the nearest neighbor, the algorithm can be extended to find the closest k nodes for any k merely by repeating calls to GetNext. Further, it can be used to find all nodes within distance r. The cost of these changes depends on the most distant node returned. Thus, we can use these techniques to fill the neighbor table efficiently.

4.4 A Dynamic Algorithm

The algorithm can be easily extended to allow nodes to join at the lowest level. In the process of finding its nearest neighbor, a node finds its nearest level 1 node, and that node becomes it parent. However, a truly dynamic algorithm must allow the join at any level, and a level i node needs to find its level $i-1$ children, and the data structure we presented does not make this easy, because children can be arbitrarily far away from their parents. In expectation, the distance is small, but is not bounded. Krauthgamer and Lee get around this problem because their construction ensures that parents are close to children.

To solve this problem, we require that each node write a pointer to itself on every node in its certificate. That is, if A is in the certificate of B, then A has a pointer to B. Further, when A gets a new child, it notifies B.[1] Suppose a node C enters that should be a part of B's certificate. Then C's parent must also be in B's certificate. But then the parent of C has a pointer to B, and so B will be notified about C's entrance, and C will have a pointer to B.

The root has to store store pointers to all the nodes, since it is in every node's certificate. One way of dealing with this is to use load balancing similar to that described before. More precisely, have every node be the the root of some set of certificates. Each node is in the static tree for every other node, storing only its parent and children.

4.5 A more practical adaptive scheme

This section outlines a sort of hybrid of the two schemes. The question in both schemes is how many nodes to keep in step 5 of GetNextList. The algorithm of Section 4.1 keeps a fixed number, and chooses that number large enough to ensure that with high probably, no more are needed. On the other hand, Section 4.2 goes to great lengths to keep exactly those that need to be kept, and no more. Thus, the second algorithm is more efficient, but also more complex.

A simpler approach is to guess at how many need to be kept. If the guess is to low, return to that level and get more, as in Section 4.2. If the guess was too high, the algorithm does more work than it needs to. That idea motivates the sketch in this section.

More mathematically, with an upper bound the d_i's, it is easy to calculate upper bounds on the q_i's, and using these too-big q_i's, do the search as before. If this is done, we find all the nodes in the certificate, including the nearest node. Unfortunately,

[1] An alternate solution was presented in Section 4.1. That section noted that with high probability, only the closest $O(\log n)$ level-$(i-1)$ nodes are ever children. These nodes can be found as described above.

```
method GetCertificate ($x$,rootNode)
  certificate[maxLevel] $\leftarrow$ {rootNode}
  queryDist[maxLevel] $\leftarrow \infty$
  level $\leftarrow$ maxlevel - 1
  while level $> 0$
     (certificate[level], queryDist[level]) =
        GetNextList($x$, certificate[level + 1], level,
        queryDist[level + 1])
     while 3* queryDist[level] $>$ queryDist[level + 1]
           queryDist[level + 1] = 3*queryDist[level]
           level $\leftarrow$ level + 1
     level $\leftarrow$ level - 1
end GetCertificate

method GetNextList ($x$,neighborlist, level, queryDist)
  nextList $\leftarrow \emptyset$
  minDist $\leftarrow$ queryDist
  for $n \in$ neighborlist
      if (d($n$,$x$) $\leq$ queryDist
         temp $\leftarrow$ GetChildren($n$, level)
         for $m \in$ temp
             if $d(m, x) \leq$ minDist
                minDist $\leftarrow d(m, x)$
  return (nextList, minDist)
end GetNextList
```

Figure 4.7: *A different method of certificate generation.* This method does not have provable bounds, but it may be simpler and easier to implement in practice.

nodes outside the certificate may also be contacted, so the bounds on certificate size are not directly applicable.

However, because it does not use a heap, it may be simpler to implement, and would probably perform equally well in practice. A simple way to upper bound the d_i's would be to query only the closest k nodes at every level (for k some small value like one or three), and use the results to get d_i's.

4.6 Experimental results

This section gives results for the neighbor search algorithms described in this chapter. This data is generated using simple simulation in Ocaml. We ignore any network effects such as message delays or lost messages. Though a simple simulation, we can answer some interesting questions.

The graphs presented here were generated using one of two underlying metrics. One is Euclidean space, where the points are randomly chosen out of the 10,000 by 10,000 grid. The other data set, which we call the King data, is pairwise ping data from 2051 randomly chosen IP address in the Internet. This is not a metric space; while it has been artificially modified to be symmetric, it does not obey the triangle inequality. The data was collected by Dabek and Li [2], using the technique of Gummadi, Saroiu, and Gribble [GSG02].

4.6.1 Costs

Section 4.2 presented a somewhat complex nearest neighbor algorithm and showed that it need only send $O(\log n)$ messages. To do this, it first bounded the certificate size (the number of nodes who are asked for children), and then argued that the total number nodes contacted (the children of all nodes in the certificate) is only a constant factor more than the number in the certificate. In this section, we try to empirically determine if the constant in the $O(\log n)$ is reasonable. To that end, Figure 4.8 shows the certificate sizes as well as the total number of pings as a function of the digit size. [3] The graph uses the King data set. Contacting a node in the certificate requires sending an more complicated message than the standard ping, and so may consume more resources. Thus, we include both measurements of complexity. (Note that since the triangle inequality does not hold on the King data set, the algorithm may not

[2] http://www.pdos.lcs.mit.edu/p2psim/kingdata/

[3] Our notion of distance is network latency, so a distance measurement is a ping. Though a good latency estimate may require more than one ping, we equate distance measurements with ping messages.

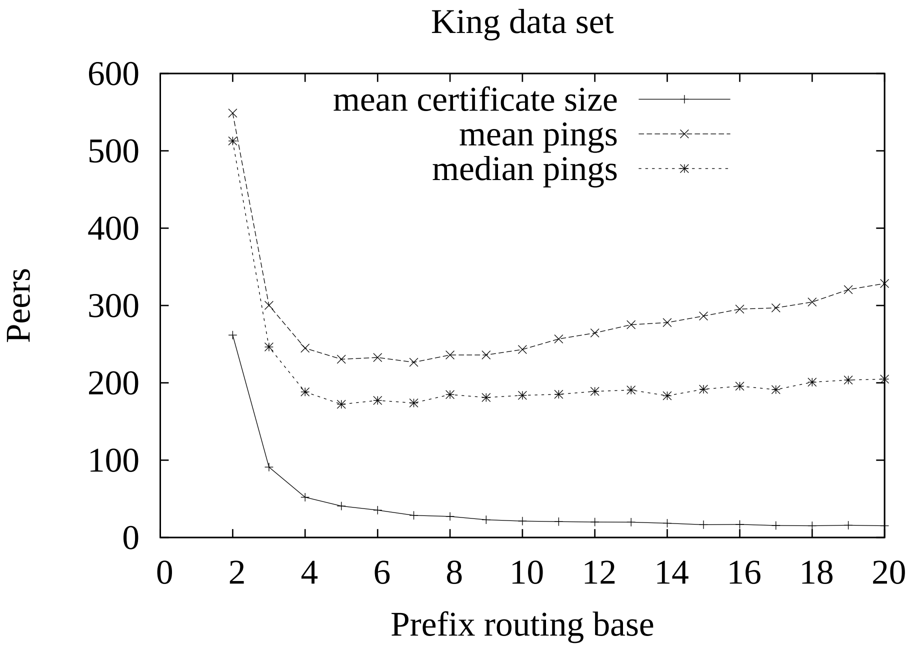

Figure 4.8: *Certificate size as a function of digit size.* We show both the certificate size and the total number of messages sent (all children of nodes in the certificate get a ping message) on the King data set. Certificate size and numbers of pings are both measures of how much work the algorithm of Section 4.2 does.

return the correct answer.)

There are two observations. First, a digit size of two is much more expensive than a digit size of three, regardless of whether the measure is total distance queries or the size of the certificate. Second, computing the full certificate requires hundreds of distance measurements (pings), no matter what the digit size, while the certificate size gets very small for large digit sizes.

This amount of work to find one entry in the neighbor table would probably not be practical. However, note that finding the certificate finds much of the neighbor table, so the work could be amortized over the entire neighbor table, which make it more reasonable.

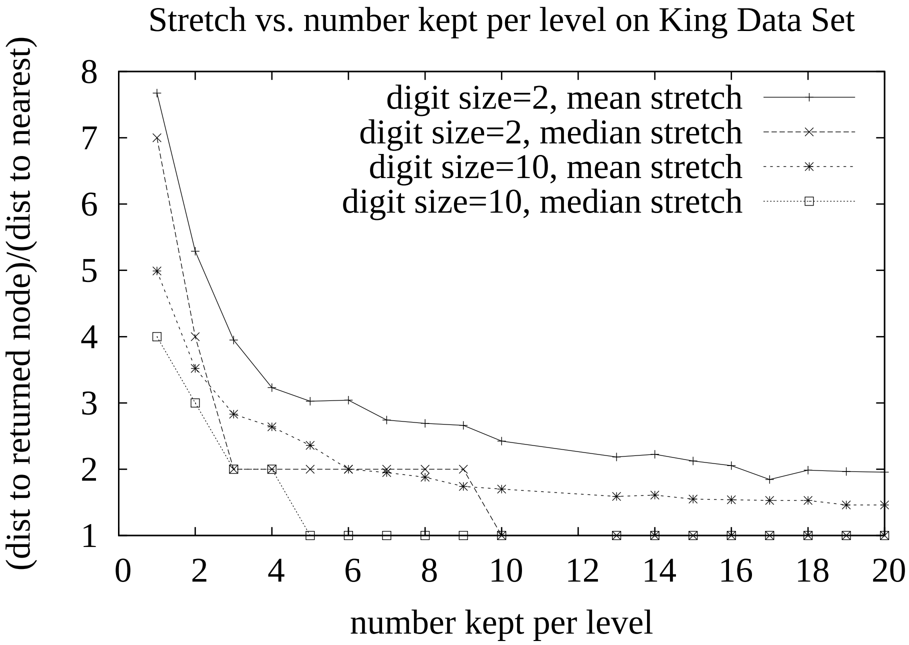

Figure 4.9: *Performance of the simple algorithm as a function of the number of nodes kept per level.* This is for the king data on 2051 nodes. Each run contains one hundred points, and the results are averaged over ten runs.

4.6.2 Correctness of Simpler Scheme

Based on the results of Figure 4.8, the algorithm of Section 4.2 is likely too costly on realistic networks. This gives rise to the question of how well the simple algorithm, which keeps a fixed number of nodes at every level, performs. The answer is mixed—in some cases, it performs well, in others, it does not. Figure 4.9 shows the performance of the algorithm, measured in terms of the ratio of the distance to the found node to the distance to the optimal node. Finally, even for small constants like three, this may require hundreds of pings, and so may be too expensive to use in practice.

Next, we compare these algorithms in a fair way to PNS(k). We pick PNS(k) because it simple, works well in practice, and is easy to control the amount of work it does, making fair comparisons possible. PNS(k) was first described by Gummadi et al. [GGG$^+$03] where they showed it worked well as part of a DHT. PNS is short for Proximity Neighbor Selection, and the k is a parameter. The algorithm works as follows. It chooses k random nodes, measures the distance from the query point to those k nodes, and returns the closest node. For example, PNS(1) chooses a node at random and returns that node, while PNS(2) picks two nodes and returns the closest. This simple scheme works very well in practice [GGG$^+$03, RGRK, RGRK03]. We chose as our measure of performance the number of distance queries or pings.

We then compare the neighbor search algorithm of presented in this chapter to PNS(k), as follows. For every test point, we run the Tapestry algorithm, counting the number of distance queries. We then allow PNS(k) to make exactly the same number of queries. The results of this comparison are shown in Figures 4.10 and 4.11.

When the Tapestry digit size is two, these two algorithms are very close in performance. Given the simplicity of PNS(k), this suggests that PNS(k) is the right choice. When the digit size is 16, the difference between the Tapestry search algorithm and PNS(k) is notably larger, but not stunningly so. Worth noting, however, that when 5 nodes are kept per level, the Tapestry algorithm returns the exactly correct answer most of the time, while the comparable PNS(k) does not perform as well.

This is consistent with the results of Rhea et al. [RGRK, RGRK03], which compares the performance of several neighbor search algorithms in Bamboo (structured similar to Tapestry) in a highly dynamic network for a different topology. It shows that Pastry-style local search, the Tapestry-style search, and PNS(k) [GGG$^+$03] all perform similarly, and if there is a best choice, it is PNS(k). Our results suggest that the performance of PNS(k) in the dynamic Bamboo network may be fundamental to the algorithms and not the result of the dynamic nature of the network.

To check the consistency of the results, we ran the same test on a 50,000 node network in Euclidean space. Figure 4.12 shows that the Tapestry neighbor search algorithm is clearly better in this context. Notice that even when the algorithm keeps only one per level, the median Tapestry stretch is 1, which means at least half the queries return the exactly correct answer! On the other hand, the comparable PNS(k) returns an answer with a much higher stretch. Thus, the network model matters a great deal.

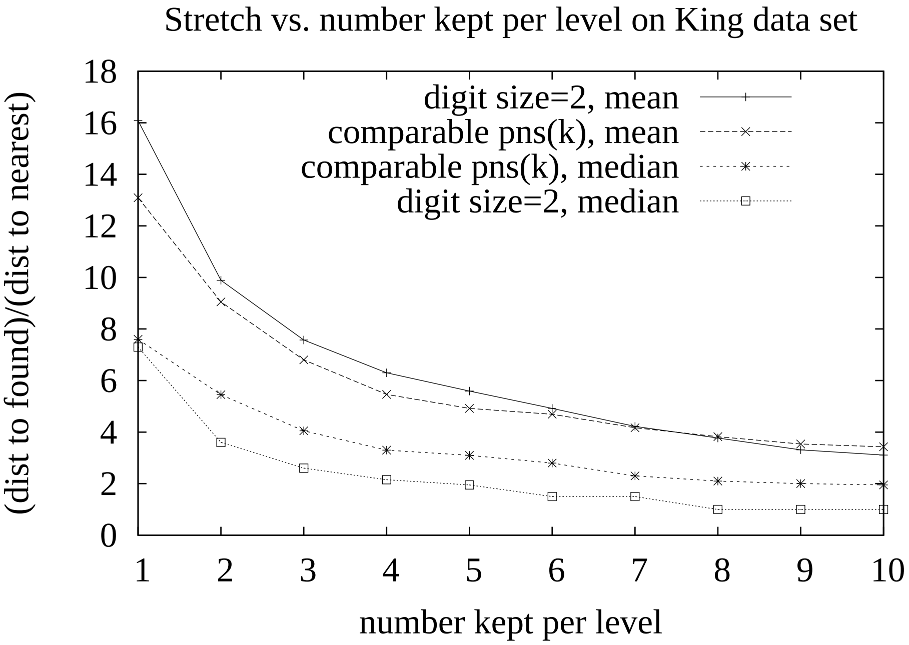

Figure 4.10: *When the digit size is 2, PNS(k) is as good as Tapestry.* The x-axis shows the number of nodes kept per level in the algorithm of Section 4.1.

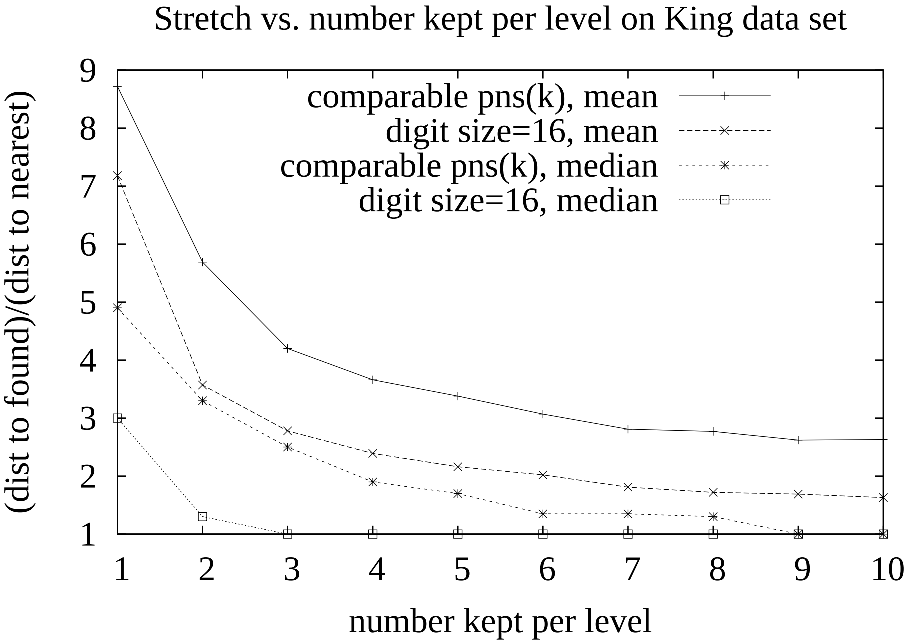

Figure 4.11: *When the digit size is 16, Tapestry is only slightly better than PNS(k).*

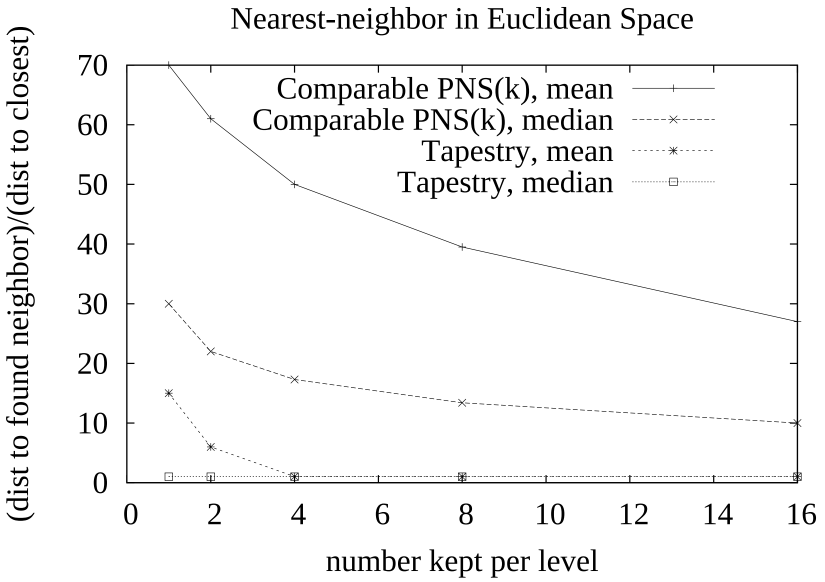

Figure 4.12: *Tapestry neighbor search vs. PNS(k) in Euclidean Space.*A comparison of the neighbor search algorithm of Section 4.1 and the comparable PNS(k). The network contains 50,000 nodes, and the search was run from 1000 random points.

This difference could be due to peculiarities of the King data set. In particular, the King data contains surprisingly large violations of the triangle inequality; it is not clear whether this is the effect of the data collection method or really represents the Internet structure. Nonetheless, the fact that [RGRK, RGRK03] reach a similar conclusion using a different network model (it uses ModelNet [VYW$^+$02]) suggests that the mediocre performance of the Tapestry neighbor algorithm compared to PNS(k) cannot be blamed on the King data set.

The two experiments (here, and Rhea et al.) cannot be directly compared, as Rhea et al. [RGRK, RGRK03] examine at the performance of nearest neighbor techniques as part of a DHT in a dynamic network, while this work focuses purely on the abstract problem of finding the nearest neighbor, but the results are still suggestive. Both the results of [RGRK, GGG$^+$03] and the King results presented here used networks in the thousands,[4] so scale may be partial explanation.

[4] Rhea et al. [RGRK] uses a network with 10,000 nodes, but it picks only 1000 to participate in the protocol.

Chapter 5

Faulty Peers

5.1 Faults and Peer-to-Peer Networks

In peer-to-peer systems, all nodes are roughly equal. This equality brings with it the potential for great power: such systems lack a central point of failure and thus could, in principle, be less vulnerable to faults and directed attacks. Unfortunately, achieving this advantage is difficult because peer-to-peer algorithms propagate information widely—greatly expanding the damage wrought by faulty or malicious nodes. In this chapter, we take a step forward by showing how two peer-to-peer systems, Pastry [RD01a] and Tapestry [ZHR$^+$03], can be made tolerant to a limited class of failures and attacks.

These systems operate properly if they successfully route messages. Moreover, they must be able to dynamically construct their routing information. We show that an additional factor of $O(\log n)$ space overhead (ie, the normal table size is $O(\log n)$, and for the fault tolerant algorithms, it needs to be $O(\log^2 n)$) they can continue to achieve both of these goals in the presence of a constant fraction nodes that do not obey the protocol. This chapter considers a model in which nodes may be faulty at the overlay level, but cannot modify messages on the wire. Also, the faulty nodes have no control over their IDs or location. While this model is weaker than one might like, it does address the very common cases of bad code and flaky hardware, and even adversaries of limited power.

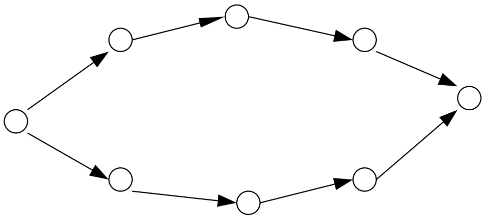

Figure 5.1: *Multi-path diversity.* If any node on the path fails, the whole path fails.

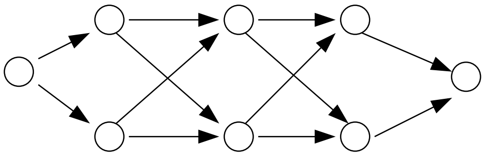

Figure 5.2: *Wide-path diversity.* Two nodes must fail at same level to break path.

The key idea of this paper is to exploit redundancy to tolerate faults, both in building the neighbor table and in routing. There are two basic approaches to fault routing illustrated in Figures 5.1 and 5.2. The first idea (in Figure 5.1) is to use multiple paths. As long as one path is failure-free, the message will make it from the source to the destination. However, notice that if one node fails in a path, the whole path is useless.

Figure 5.2 shows a different technique. Instead of two separate paths, this diagram show a single path that is two nodes wide. This means that all the nodes in a given step send to all the nodes in the next step. If any node in one step gets the message, then all the nodes in the next step will also get the message. For the routing to be blocked, at some step, both the top and the bottom nodes must *simultaneously* fail. This provides much greater fault-tolerance per redundant overlay node than multiple paths (even normalizing for bandwidth consumed). We will exploit this technique later.

There is another, orthogonal, routing design decision. The routing outlined above is *recursive*. In recursive routing, the intermediate nodes forward the query on to the next intermediate node. In contrast, some routing algorithms (including the ones described in this paper) are *iterative*. In iterative routing, at each step, the initiating node contacts some other node (or a set of nodes) to get the next hop. The difference is illustrated in Figure 5.3. Iterative routing is less efficient, but gives the originating node more control, which can be important in a faulty network.

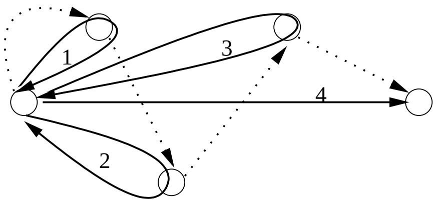

Figure 5.3: *Iterative vs Recursive routing.* The source is on the left, the destination is on the right. The solid arrows, labeled by number, represent the iterative path, and the dotted arrows represent the corresponding recursive path.

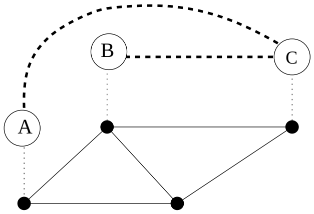

Figure 5.4: *Overlay Network.* Overlay nodes (hollow circles) with neighbor links (dashed lines) above physical nodes (solid circles) and network links (solid lines). B cannot interfere with messages from A to C, but C can interfere with messages from A to B.

5.1.1 Related Work

Sit and Morris [SM02] identify some basic categories of attacks on peer-to-peer networks and some general responses to them. Their suggested responses are general enough to apply to most peer-to-peer networks, so they are in some cases not completely specified. Thus, they do not formally prove that their approaches succeed. Two of their concerns are the two problems addressed in this paper—attacks on dynamically building of the routing tables and routing itself. They also suggest that iterative routing may be better in faulty networks.

Douceur in [Dou02] describes the Sybil attack, in which a faulty node generates many IDs and then pretends to be many nodes. He shows that it is practically impossible to prevent this attack without a centralized authority which either explicitly or implicitly distributes IDs. (An IP address is an example of an implicit identifier.)

Castro *et al.* [CDG$^+$02] address many the same issues as the Sit and Morris, but as they relate to Pastry in particular. For example, to route securely, Castro *et al.* first route normally, and then perform a routing failure test to determine whether the routing has gone wrong. If their test detects that routing may not have been correct, they retry routing with a secure routing protocol.

Their secure routing protocol uses a different data structure that, unlike the normal routing table, does not take into account network distance. With the assumption that nodes cannot choose their location in the network, and an algorithm to build the normal table correctly with high probability, [1] this backup table may be less important.

Their secure routing technique is essentially that of Figure 5.1. That is, they send a message from r different starting points. But this technique requires r to be polynomial in n to get a failure probability (over all paths) to be less than a constant. This paper introduces a different technique, using the "wide" paths of Figure 5.2, that can gives a failure probability of $1/n^c$ when r is only $O(\log n)$ (or $O(\log^2 n)$ in an stronger fault model).

Saia *et al.* [SFG$^+$02] and Naor and Wieder[NW03b] also made use of the wide paths depicted in Figure 5.2. In both cases, they group together nodes into groups of size $O(\log n)$ and use those groups as a single virtual node in the overlay. The network routes correctly as long as one node in each unit follows the protocol. Their routing algorithms are recursive, rather than iterative.

Fiat and Saia [FS02] construct a butterfly network of virtual nodes where each virtual node is made up of $O(\log n)$ real nodes. (In [SFG$^+$02] they make this construction dynamic.) By having more than one starting point, even when the failures are picked to be the worst possible, their system still performs well for almost all objects and almost all searchers. In their system, nodes have degree $O(\log^2 n)$. Naor and Wieder take a different approach. They build a constant-degree DHT (detailed in [NW03a]), and then force each node to act for $O(\log n)$ others. Since the original degree was constant, the final degree is still a reasonable $O(\log n)$.

Section 5.3 gives similar techniques that can be used to "retrofit" Pastry and Tapestry with fault tolerance. These techniques may be useful even with constant overhead, and in Section 5.5 shows some simple experiments that suggests this.

[1] With high probability means the probability of failure to be less than $1/n^c$ for $k = c' \log n$, for some c' that depends on c.

5.1.2 The Main Idea

We address two problems. The first is how to build the data structures necessary for routing in the presence of faulty nodes. The second is how to route securely. As mentioned above, we discuss Pastry [RD01a] and Tapestry [ZHR$^+$03].

This chapter considers a tree on the network (with the root at the destination), and this routing process involves traveling from the leaf to the root. (A different destination gives a different tree on the network.) At each routing step, the message travels to some node that is closer to the root in the ID space (that is, the a node with a longer ID match). Generally, there are many such nodes, and the systems choose the node that is closest in terms of network distance. This gives a short path, not just in terms of overlay hops, but also in terms of network distance. However, if the path uses one bad node, the message may never reach the root, or it may reach the wrong root. Since there are $O(\log n)$ hops, the probability that at least one of them is faulty is quite large.

The solution in both cases is avoid relying on a single source and to use measurements of network distance to decide what information is valid.

Section 5.4 looks at the difficulty of building a correct table when nodes send incorrect information. In Pastry and Tapestry, nodes are given freedom to choose from among a set of choices. This flexibility gives better performance. However, both [SM02] and [CDG$^+$02] argue that this flexibility is harmful because it means that a node has a difficult time determining what information can be trusted.

Section 2.1 presents two routing techniques similar in spirit to that of [SFG$^+$02, NW03b] in that they route to a set of nodes at each level instead of just one. However, in this paper, these sets are not determined based on IDs, but on network distance. At each step, the algorithm queries the nodes in the current set to get the next set. If one node in the current set is good, the hope is that at least one node in the next step is also good. There is a subtlety here. If the nodes in the current set return do not return the same set of nodes for the next step, the next set could be larger than

	Extra Work		
Scheme	fail-stop	bad-data	additional assumption?
Neighbor Table (Section 5.4)	$O(l)$	$O(l^2)$	—
Routing I (Section 5.3.1)	$O(l)$	$O(l^2)$	fraction bad nodes $< 1/c^2$
Routing II (Section 5.3.2)	$O(l^2)$	$O(l^2)$	—

Table 5.1: Summary of paper results: All algorithms assume that $c^2 < b$, and that l, the measure of redundancy, is $\Omega(\log n)$ and chosen sufficiently large that with high probability, one node in l is good. When nodes are malicious, we assume the network is growth-restricted.

the current set. This growth in the set size means that this technique could amount to flooding the network, which would be very inefficient.

However, if the algorithm eliminate some next set possibilities, there is a danger it will eliminate the only good possibilities. (And if failed nodes tend to return other failed nodes, the percentage of failed nodes may be higher than the percentage of failed nodes in the network as a whole.) The algorithm must ensure that it always keeps at least one good possibility for the next step. Our idea is to use the network distance from the query originator to determine which nodes are in the next set, since in our model, the adversary cannot manipulate the network distance. For a small enough fraction of failing nodes, this succeeds with high probability when the sets are of size about $O(\log n)$. Section 2.1 presents two algorithms implementing this general idea. Both results rely on a restricted-growth assumption about the network, but these techniques may still perform well in other situations.

We show that the tables used by Tapestry and Pastry can be built correctly even in the presence of faulty nodes by using a factor of $O(\log n)$ additional storage, where n is the number of nodes in the network. (This gives a total storage of $O(\log^2 n)$.) The key idea is to gather a list of candidates for a given slot from enough sources that the probability all are faulty is low. Then we can use the network distance to determine which candidate is the best.

Table 5.1 summarizes our results.

5.1.3 Model

Defining an appropriate failure mode is a difficult task. A weak model may not be realistic enough, and a model that is too strong is impossible. Our goal is to present a model that is strong enough to deal with at least some real problems, but that still allows analysis.

To explain our model, it helps to view the network as made up of two layers. The underlying layer can be trusted to deliver messages. The upper layer, or overlay layer, consists of the peer-to-peer nodes and the connections they maintain with each other (using the underlying layer for routing). This overlay is not trusted. These nodes can misbehave, but they cannot destroy or modify message on the wire, only messages that go through the overlay layer. Figure 5.4 shows this distinction. Furthermore, this paper makes the following assumption about overlay nodes:

- Node IDs are assigned securely.
- Nodes fail, or become corrupted, independent of their location.
- The network must form a restricted-growth metric space. Let $\mathcal{B}_r(A)$ be the ball of radius r around A, or all the points within r of A. Then the condition needed is

$$|\mathcal{B}_{2r}(A)| \leq c\,|\mathcal{B}_r(A)|\,, \tag{5.1}$$

  where c the expansion constant of the network. This assumption need not hold in the fail-stop model.
- Part of the trusted component is the ability to measure network distance(i.e., by pinging).
- For one of the secure routing algorithms (Section 5.3.1), we assume the fraction of bad nodes is small compared to the growth rate of the graph.

All our results apply to the fail-stop model, where nodes that fail simply cease responding to messages, but do not misbehave. However, they also apply in a worse case, in which "failed" nodes send messages with bad data.

The major difference between our model and that of Castro *et al.* [CDG+02] is focused around our assumptions on the network topology. This paper requires a secure way of measuring network distance, supplemented with the assumption that nodes fail independently of their distance. While these requirements may seem limiting, note that changing distance measurements, particularly making them shorter, is actually quite difficult without interrupting the flow of messages at the lowest level—a capability for denial of service that is outside the scope of this paper (and of the Castro *et al.* solution).

Second, relaxation of the location independence requirement can lead to routing failure in regions with high local concentrations of bad nodes; in this circumstance, our techniques could be supplemented with any non-local technique (for example [CDG+02, NW03b, SFG+02]).

5.2 Some preliminary results

First, we state without proof a lemma. (The proof is very similar to that of Lemma 5 from Chapter 4.) Suppose a ball around A of radius r contains k nodes at level-i. The lemma bounds the probability that a larger ball around A (of size $3r$) contains k level-i nodes that also satisfy predicate $p(X)$ where nodes satisfy $p(X)$ independently with probability f.

Lemma 13. *Suppose a ball of radius r contains $k = O(\log n)$ level-i nodes. Then so long as $fc^2 < 1$, the ball of radius $3r$ contains no more than k level-i nodes satisfying p (where c is the expansion constant of the network.)*

The special case when $p(X)$ is the probability that X is a level-$(i+1)$ node will turn out to be particularly useful, so we formally state it in Corollary 1. This lemma and the remainder of the paper require that $c^2 < b$.

Corrolary 1. *Suppose the inner ball of radius r contains $k = O(\log n)$ level-i nodes. Then so long as $c^2/b < 1$, the ball of radius $3r$ contains no more than k level-$(i+1)$ nodes.*

5.3 Fault Tolerant Routing

In [CDG$^+$02], the authors present a routing technique for dealing with hostile networks. Their idea is to use r different paths to the root, and then if a fraction f of the nodes are corrupted, then the probability a message reaches the destination along a particular path is $(1-f)^{1+\log_b n}$. The probability that all r of the paths fail is $((1-f)^{1+\log_b n})^r$. Asymptotically, this is a low probability of success. In fact, using their formulation, the probability of failure is more than $(1-(1-f)^{1+\log_b n})^r$, which is approximately $\exp(-rn^{\frac{\ln(1-f)}{\ln b}})$. If the desired failure probability is constant, then r must be a polynomial in n.

We give two techniques for more fault tolerant routing. Both use iterative routing. Recall that in iterative routing, the initiating node controls the process. Given a list of ith hop nodes, the node wanting to route contacts one of them asks for possible $i+1$st hops.

If an ith hop node returns nodes that look like good $(i+1)$st hops but are not, then the initiating node does not know which of the nodes are good. In the fail-stop model, the initiating node can just try one of the next-hop nodes. If the initiating node gets no reply, it tries another. But consider the case where the nodes are malicious but of limited power and want to pass messages on to a particular subnet (perhaps their own, because they control it, or perhaps another to get rid of traffic). Picking from among the returned nodes at random is a problem, since the misconfigured nodes may be more likely to return other misconfigured nodes, while the correct nodes may return misconfigured nodes with probability proportional to the fraction of misconfigured nodes. Both of the techniques of this section attempt to deal with this case by using network distance information to pick from this set of returned nodes ones that are reasonably likely to be good.

Assume that the fraction of faulty nodes is f. Both of our techniques require each node to store not one neighbor in each entry in the routing table, but $l = O(\log n)$. In the tree terminology, this means each node stores not one parent, but l parents. (The resulting structure is no longer a tree, but the parent and child terminology still applies.)

The first technique is simpler and more practical algorithm, but the analysis is more complicated and holds only holds when f is small.

5.3.1 Routing Technique I

The algorithm works as follows.

- The query node starts with a list of l level-i nodes. It then contacts each of these nodes and asks for the their l closest level-$(i+1)$ nodes.
- The query node then gets a list of nodes, eliminates duplicates, and measures the distance to all of them. It then chooses the closest l, and goes back to the first step.

At some point, there will be no nodes at the next level, and the algorithm has found the root. To prove this works, we need to show is that at each step, there is at least one good node among the l.

Let the query node be Q. We start by proving the following lemma relating distance to a node to the distance to its parent.

Lemma 14 (Circle Lemma). *Let r be a radius such that $\mathcal{B}_r(Q)$ (the ball of radius r around Q) contains at least l level-$(i+1)$ nodes. Then for any level-i node within r of Q, all its parents are within distance $3r$ of Q.*

Proof. See Figure 5.5. Suppose B is a level-i node within distance r of Q. Note that B has l potential parents within r of Q. By the triangle inequality, all these nodes

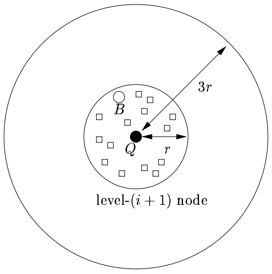

Figure 5.5: The parents of B lie within the big circle. The squares represent some of the level-$(i+1)$ nodes that B could choose as parents.

are within $2r$ of B. If B chooses different parents than the nodes within the smaller ball, it could only be because they were closer, so all of B's parents are within $2r$ of B. But this means they are within $3r$ of Q. □ □

Now, let r_i be the radius of the smallest ball around Q containing l level-i nodes (r_{i+1} is defined similarly). By Lemma 1, $3r_i < r_{i+1}$ with high probability. Combining that with the Circle Lemma, gives

Corrolary 2. *Consider a set of l level-i nodes within r_{i+1} of Q. Suppose $fc^2 < 1$ (that is, the fraction of bad nodes is sufficiently low) and that there is at least one good node among the l nodes. Then with high probability, we will be able to find l level-$(i+1)$ nodes within r_{i+2} such that at least one of them is good.*

Proof. Consider the ball of r_{i+1} around Q. We know that there is at least one good level-i node inside r_{i+1}. Call this node B. By the Circle Lemma, B's parents are all within $3r_{i+1}$, and by Lemma 1, $3r_{i+1} < r_{i+2}$. Then B's most distant parent gives us a bound on the distance to the furthest node in the next level list.

Using Lemma 13, the number of bad level-$(i+1)$ nodes within $3r_{i+1}$ is less than c^2fl with high probability for $l = O(\log n)$. So if $fc^2 < 1$, there are not enough bad nodes within the bound given by B's parent, so of the l level-$(i+1)$ nodes, at least one of them is good. □ □

Note that this is a pessimistic proof. Even if there are l bad nodes close enough to the query node, it is not immediately clear how bad nodes could take advantage of that fact without a great deal of coordination.

This algorithm does l^2 extra work in the worst case, since each of these l nodes could return l different parents, giving a total of l^2 answers, each of which must be contacted. In the fail-stop case, where nodes do not return bad data but simply stop working, most of these l^2 nodes are the same, so the algorithm need only ping $O(l)$. To see this, consider the level-i list. All those nodes are within r_{i+1}, and all their parents will be within r_{i+2}. By the Equation 5.1, the number of $i+1$ nodes within r_{i+2} is expected to be c^2l.

5.3.2 Routing Technique II

This algorithm gives a tighter bound in the proof and works for any value of f (by picking l large enough), independent of the network, but this would not be as convenient to implement in practice.

At every step, this algorithm ensures that it knows the closest l level-i nodes. In Section 5.3.1, the algorithm did not guarantee that it had the "closest" l nodes. Knowing that they are the closest means they are determined by the structure of the network and not by the misbehavior of the bad nodes, so the probability of failures is independent (given our network assumptions) among these l nodes. The algorithm works as follows:

1. From the level-i nodes, pick all the nodes that are also level-$(i+2)$ nodes. In Lemma 15, we show that $l = O(\log n)$ is big enough such that with high probability, at least one of the level-i nodes is good level-$(i+2)$ node.

2. Get the children of these nodes.

3. Pick the closest l of these children to be the set of level-$(i+1)$ nodes.

To prove this works, we show two things. First, that the first step succeeds; that is, that there is at least one good level-$(i+2)$ node among the l level-i nodes. Second, we show that if there is such a node, it will be able to return all good children. This is true if the closest l level-$(i+1)$ nodes all have the level-$(i+2)$ nodes from the first step as parents. The first step is shown next.

Lemma 15. *For $l \geq \log(1/\epsilon)\frac{b^2}{1-f}$, the probability than none of the level-i nodes are good level-$(i+2)$ nodes is less than ϵ.*

Proof. Given a level-i node chosen uniformly at random, the probability it is a level-$(i+2)$ node is $1/b^2$. Since failures are independent of node ID and location, the l closest level-i nodes are independent trials. The probability a node is good is $(1-f)$, so the probability is it a level-$(i+2)$ node and not faulty is $\frac{(1-f)}{b^2}$. Given l level-i nodes, the probability that none of them are good level-$(i+2)$ nodes is $(1-\frac{(1-f)}{b^2})^l \leq \exp(-\frac{l(1-f)}{b^2})$, and picking $l = \log(1/\epsilon)\frac{b^2}{1-f}$, the probability the list does not contain a suitable level-$(i+2)$ node is less than ϵ. □ □

Setting $\epsilon = 1/n^c$, the bound is a high probability bound. Next, we show the second part, that the level-$(i+2)$ nodes chosen have as children the closest l level-$(i+1)$ nodes.

Lemma 16. *Suppose A is a level-$(i+2)$ node and is among the closest l level-i nodes to B. Then for $l = O(\log n)$, with high probability, the closest l level-$(i+1)$ nodes to B point to A.*

Proof. We prove this using Corollary 1. As usual, let r_{i+1} be the smallest radius such that the ball of radius r_{i+1} around B contains l level-$(i+1)$ nodes.

Consider a level-$(i+1)$ node, call it C, within r_{i+1}. Notice that C has a potential parent, A, within distance $r_{i+1} + r_i \leq 2r_{i+1}$. If it does not have A as a parent, then it is the case that there are more than l level-$(i+2)$ nodes within $2r_{i+1}$ of C. But since $d(B, C) \leq r_{i+1}$, this implies that there are at most l level-$(i+2)$ nodes within $3r_{i+1}$ of B. Now apply Corollary 1 to say that this is unlikely when $c^2 < b$ and l is chosen large enough. □ □

```
method FindNearestNeighbor (QueryNode, RootSet)
  $maxLevel \leftarrow$ Level(RootSet)
  list $\leftarrow$ RootSet
  for $i = maxlevel - 1$ to $0$
      list $\leftarrow$ GetNextList(list, $i$, QueryNode)
end FindNearestNeighbor

method GetNextList (neighborlist, level, QueryNode)
  nextList $\leftarrow \emptyset$
  for $n \in$ neighborlist
      temp $\leftarrow$ GetChildren($n$, level))
      nextList $\leftarrow$ KeepClosestk(temp $\cup$ nextList)
  return nextList
end GetNextList
```

Figure 5.6: Finding the Nearest Neighbor: This algorithm operates with respect to the tree defined by the **RootSet**. For more detail, see text.

5.4 Fault Tolerant Neighbor Search

Recall that for a $O(b \log_b n)$ of different prefixes, a node must store the *closest*, in terms of network distance, node with that prefix. When nodes are inserted, they must also be able to find the closest node with the given prefix. The algorithm of the section gives a way to do this.

As mentioned before, viewing the algorithm as a black box would require running it once for every prefix. In fact, it can also be used more cleverly. By choosing the appropriate starting point, in one pass it finds one entry at every level. Further, [HKRZ02] showed that by choosing parameters carefully, we can use this technique to fill the entire table.

5.4.1 The Robust Algorithm

The original algorithm, described in Chapter 4, starts with a list of the k closest nodes at *maxLevel*. We call this list the **RootSet**. Assume that these are all the nodes at *maxLevel* or the closest k such nodes. Then, to go from the level-$(i+1)$ list to the level-i list, query each node on the level-$(i+1)$ list for all the level-i nodes they know, or, in other words, their children. The algorithm trims this list, keeping only the closest k nodes. The proof that the k nodes kept at each step are actually the closest nodes at that level is deferred to the next section.

The existence of one failed node can cause the algorithm to return a wrong answer. In particular, if the nearest neighbor is B, and the parent of B does not respond or does not report the existence of B, then the querying node never finds out about a B. This fragility is clearly undesirable.

This problem is solved without changing the algorithm, by changing the definition of "parent". Each level-i node B, (where level-i is defined with respect to a particular tree), now finds the closest l level-$(i+1)$ nodes and treats them as its parents, and it, in turn, is a child of all those l nodes. Notice that if *any* parent reports B, the querying node will be able to find B.

With l parents, and a failure probability of f, the probability that all the parents fail is f^l (here, failure means the node fails to return B, either because it fails to reply to the message or it returns a list without B). If $l = O(\log n)$ and f is constant, then the probability of a failure is an inverse polynomial in n. (Unless the root is assumed to be good, the **RootSet** must also be of size l. If this is not given to the algorithm, we can use the fault tolerant routing described in Section 2.1 to find a set of nodes at the right level.)

In order for this argument to hold, the algorithm must query all of B's parents. The following argument shows that this can be done with k still $O(\log n)$ (so long as $l = O(\log n)$). The idea is that any nearby node has its parents nearby as well, for the correct definitions of nearby.

Lemma 17. *For $k > bl$, with $l = \Omega(\log n)$, with high probability, the k closest level-$(i+1)$ nodes contain all the parents of the closest k level-i nodes.*

Pick $k > bl$. Then the expected number of level-$(i+1)$ nodes in a set of k level-i nodes is at least l, and if $k = O(\log n)$, we can say that the number of level-$(i+1)$ nodes is at least l with high probability. Now let r_i be the radius of the ball containing k level-i nodes. We just argued that this ball also contains l level-$(i+1)$ nodes. But now apply the Circle Lemma (Lemma 14) to say that if B is within the radius of the closest l level-$(i+1)$ nodes, then it is also with r_i. That means its parents are within $3r_i$, and by Lemma 1, $3r_i < r_{i+1}$. So with high probability, the k closest level-$(i+1)$ nodes contain all the parents of any level-i node within distance r_i. □

Since this is a high probability result, we can use the union bound to argue that over even $\log n$ levels, with high probability, there is no failure at any level.

5.5 Experiments

We implemented the nearest neighbor algorithm of Section 5.4 and the routing algorithm of Section 5.3.1. The simulation used 50,000 nodes, using a base of 10 (this means the number of steps to the root was five or six.) The underlying topology used was a grid, where the overlay points were chosen uniformly at random from a 10,000 by 10,000 grid.

The failed nodes in both cases only return information about other failed nodes. This is worse than the fail-stop model, since here nodes are actually getting bad data, but other attacks may be possible.

For varying fractions of bad nodes, we ran the nearest neighbor algorithm, and calculated the number of times that algorithm gave the incorrect answer because of the failed nodes. [2] See Figure 5.7. Notice that when half the nodes bad, the number of incorrect (i.e., not closest) nodes actually decreases; this is because returning only other bad nodes does not cause problems if the the end answer is also a bad node.

There was large variance; if nodes near the root had failed, the number of incorrect nodes returned was quite high. The situation was particularly bad when one parent is used, so no data point was included on the graph.

Figure 5.8 shows that chance of reaching the root improves a great deal with only a little additional overhead. Note that Tapestry already stores two backups for every entry, so $l = 3$ requires no additional overhead. (And the nearest neighbor algorithm

[2] The algorithm implemented here is the one described in Chapter 4, Section 4.2.

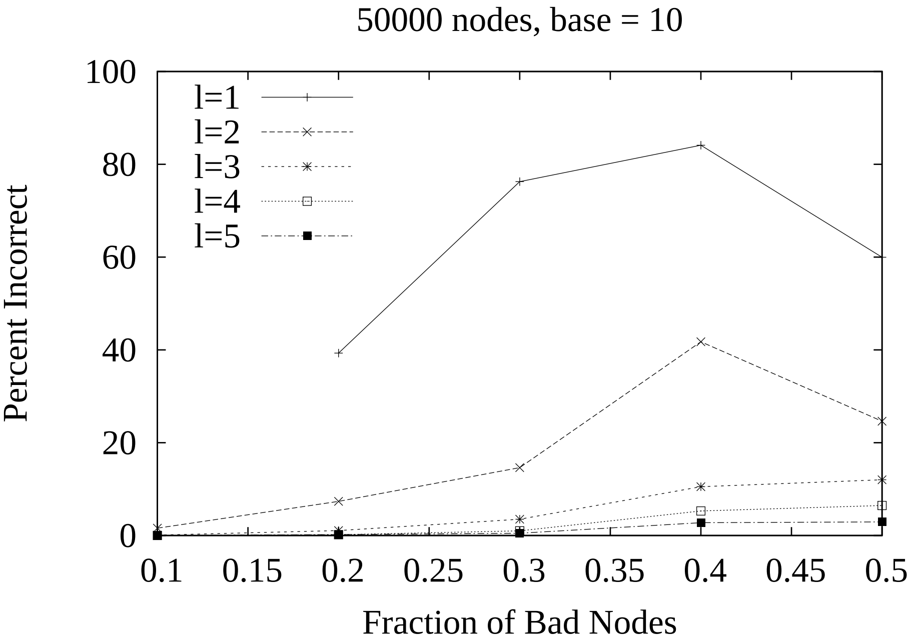

Figure 5.7: *Neighbor search with failures.* Percentage of incorrect entries over 5,000 trials in a network of 50,000 nodes. Even a small amount of redundancy (shown here as l) significantly reduces the number of incorrect entries.

uses these backups.)

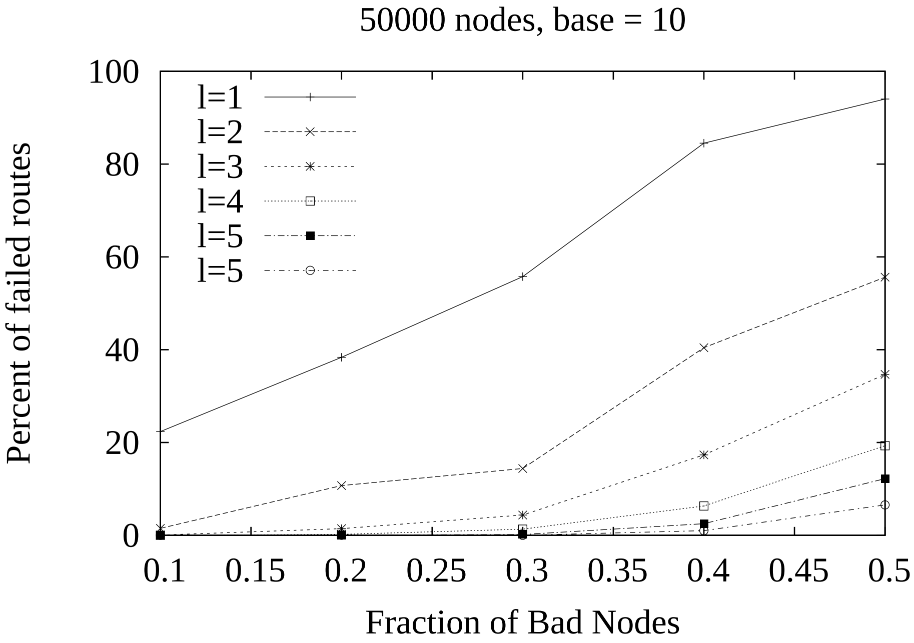

Figure 5.8: *Routing with failures.* Percentage of routes that fail to reach the root when the algorithm of Section 5.3.1 is used. Notice that a small amount of redundancy (shown here as l) helps tremendously.

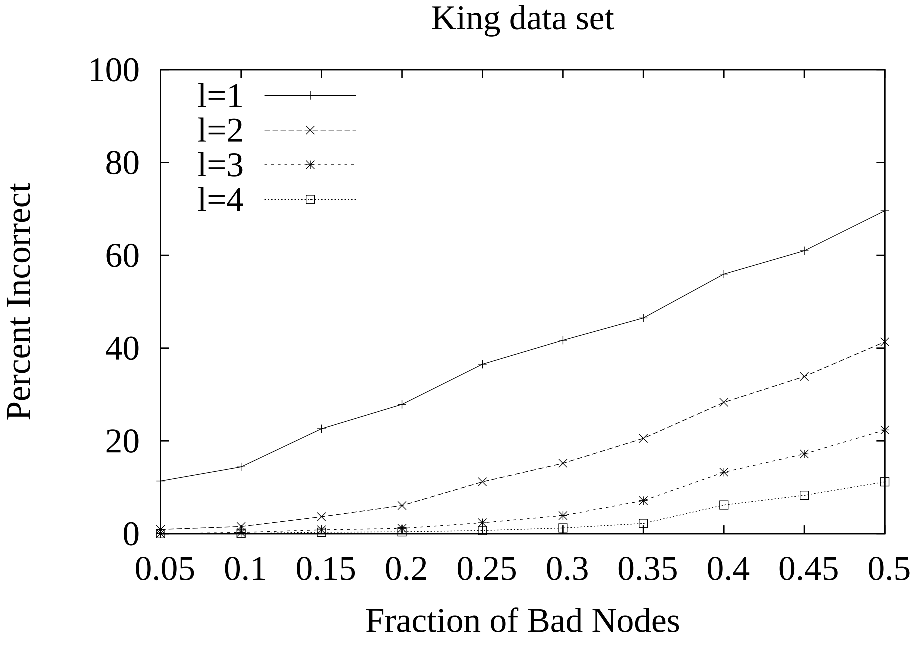

Figure 5.9: *Using Data from Internet.* This graph is the same as Figure 5.8 except that the underlying network is given by measurements of the real network.

Chapter 6

Object Location in General Networks

Until our work, there was no low-stretch object location system for dynamic networks that did not require the corresponding metric space to be uniformly growth-restricted. This chapter bridges this gap in two ways. One result (joint work with Krauthgamer and Kubiatowicz in [HKK04]) is to give a dynamic scheme for a wider class of metrics than considered in the past.

Before doing that, however, we show that the object location scheme describe in Chapter 2 is connected to the general metric space results, but showing how a version of the scheme gives low stretch in general metrics. (It is not practical for peer-to-peer networks because it is neigher self-organizing or load balanced.)

Schemes with stretch guarantees divide into two categories. Those for general metrics, and practical schemes. In growth-restricted spaces, the PRR [PRR97] scheme, and later LAND [AMD04], achieve constant expected stretch.

Outside the context of peer-to-peer networks, several structures have been proposed for object location in general metric spaces. See for example, [AP91] and [RRVV01]. The distance oracles of [TZ01] and the compact routing schemes of [ACL$^+$03b, ACL03a] also address this problem, though they don't phrase their results in this context. These schemes achieve polylogarithmic space and polylogarithmic stretch. However, these structures are impractical in the peer-to-peer world because they require the underlying network to be fixed from the start. In addition, some cannot be constructed in a distributed fashion, and many are not even load-balanced, requiring a central server.

6.1 A non-practical algorithm for general networks

This section, gives a non-dynamic scheme for an arbitrary metric space S. We show how to route to an object with polylogarithmic stretch and $O(|ID| \log^2 n)$ average space, where $|ID|$ is the size of an object ID. We remark that this is the strawman scheme proposed by Plaxton, Rajaraman, and Richa [PRR97] without load balancing, and is similar to the scheme of Thorup and Zwick [TZ01]. The proof is reminiscent of the metric embedding results of Bourgain [Bou85], and Linial, London, and Rabinovich [LLR94].

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 key to showing stretch bounds.

Theorem 6. *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 $\frac{1}{2}$ 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. And if there is $s \in S_{i,j}$ that points to both X and Y, when X queries s, X will get a pointer to Y, so $i^* = i$.

We 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

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

But this can happen at most $\log n$ times, since $|\mathcal{B}_X(r_1) \cap \mathcal{B}_Y(r_1)| \geq 2$ (since it contains X and Y) and the network has only n nodes.

Finally, if at any point $|\mathcal{B}_X(r_k) \cup \mathcal{B}_Y(r_k)|$ contains the whole network, then let $i^* = 0$, and since there is only one element of each $S_{0,0}$, it will clearly be pointed to by X and have a pointer to Y. □

To get the stretch bound, notice that if $d(S_{i*,j}, X) \leq d(X, Y) \log n$, 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 efficiently maintain this data structure.

6.2 Changing c

All the existing practical low-stretch schemes intrinsically rely on knowing a *global* growth rate upper bound C. These systems are based on assigning node IDs that are represented in some base $B \geq C$, and routing messages toward a given destination ID by fixing one digit at a time. It follows that the number of neighbors each node has to maintain is (at least) proportional to C. Since a larger number of neighbors immediately increases the amount of maintenance traffic, it is crucial for deployment that C is assigned a modest value.

In practice, however, it is likely that in many regions the global value C would be much larger than the actual local value,

$$\rho_{v,r} := \frac{|\mathcal{B}_v(2r)|}{|\mathcal{B}_v(r)|},$$

which we call the *local growth rate* around node v at scale $r > 0$. This would be consistent, for example, with the transit-stub model [ZCB96] of the Internet, which features *stub domains* (modeling local networks) and *transit domains* (modeling backbone networks that interconnect stub domains); see Figure 6.1 for an illustration. Traffic between two nodes within a domain is routed inside the domain, and is typically fast compared to interdomain links. Each stub domain probably has an internal structure, but we cannot expect that the Berkeley stub domain would be similar to the IBM Almaden one. It is therefore only natural to require that the neighbors selection policy in the Berkeley stub domain is kept independent of the internal topology of unrelated domains.

A partial remedy is to set C ignoring a few regions of extremely high local growth rate, giving up on low stretch in those regions. But even then significant gaps between the local growth rate and C might occur for the following reasons.

- The local growth rate is likely to vary significantly between different subnetworks—a local network at a small liberal arts college is probably very different from the one at a large research university.
- The local growth rate probably varies significantly between scales, because networks of different scale (e.g., local area networks vs. a nationwide network) have different technologies.
- Systems based on a global value C are designed with the value of C being determined before the network is first launched. In a dynamic system, this means that C must be increased by a large safety margin to guarantee future performance.

Large gaps between the local growth rate $\rho_{v,r}$ and the global value C might have a tremendous impact on the overall network. Many low-provisioned subnetworks would probably suffer from excessive maintenance traffic, resulting in a high network stress and poor network utilization.

It should be noted that low-stretch guarantees might require some dependency on $\rho_{v,r}$. Consider for example a subset S of k nodes, every two of which are at the same distance 1, and suppose all other nodes are at least ϵ away from every node of S. Then, a lookup whose source and destination are in S can achieve $1+\epsilon$ stretch only if it is performed in one hop, which means that every node $v \in S$ must have at least $k-1$ neighbors, where $k \simeq \rho_{v,\gamma}$ for any $\frac{1}{2} < \gamma < 1$. Notice that this scenario is

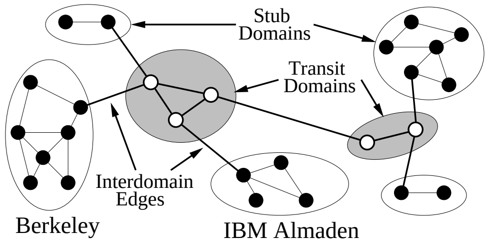

Figure 6.1: *An illustration of a (non-hierarchical) transit-stub graph.*

likely to happen in large local area networks. See also [KL04b, KL04a] for a related notion and similar lower bounds.

6.2.1 A low stretch scheme for more metrics

We present a scheme that achieves a *guaranteed* low stretch object location without requiring a global growth rate bound C, and is thus effective in a wider and more realistic class of networks than previous schemes. The distinctive feature of our scheme is that it is inherently adaptive to the underlying topology, as its operations in any locale depend on the locale's properties. In particular, our system achieves the same $1 + \epsilon$ stretch as LAND [AMD04], with a neighbor list size does not depend on C, but only on the local growth rate $\rho_{v,r}$, on ϵ, on the *normalized diameter* of the network $D_N = D/d_{\min}$, where D is the largest distance between two points in the network and $d_{\min}$ are the largest and smallest pairwise distances in the network, respectively), and on the following bounds on the rate of change of $\rho_{v,r}$:

$$\delta_s := \sup \left\{ \frac{\rho_{v,r}}{\rho_{v,2r}} : v \in X, r > 0 \right\},$$

which upper bounds the change in growth rate over different but nearby scales, and

$$\delta_d := \sup \left\{ \frac{\rho_{v,r}}{\rho_{u,r}} : u, v \in X, r > 0, \text{ and } d(u,v) \leq r \right\},$$

which upper bounds the change in growth rate over different, but nearby (relative to the scale in question), nodes.

Theorem 7. *There exists a randomized scheme that achieves object location with* $1+\epsilon$ *stretch, such that the expected number of neighbors of each node* v *is at most* $f(\max_{v,r} \rho_{v,r}, \delta_s, \delta_d, \epsilon) \cdot \log^2 D_N$*, for some function* f*.*

Our system design takes a direct approach toward achieving low stretch. As a byproduct, we get the following advantages (over existing schemes), which we believe will culminate in improved and more robust deployments of low stretch object location schemes.

- The system is adaptive to the local growth rate, which may be significantly smaller than the global upper bound C, thereby using less resources in many regions of the network.
- It operates without requiring a predetermined value C and it is therefore more practical in *any* underlying topology, since a single value of C might be hard to find. In fact, our approach identifies the algorithmic problem of estimating the distribution of the latencies between a node v and all other nodes in the network, and such an algorithm is sketched in Section 6.6.2.
- Our construction is robust to small errors in network measurements. Since accurate network measurements are difficult to make and require lots of bandwidth, this is important. While a node needs to count the number of nodes within a certain distance from it, having this number off by a factor of two will have only a slight effect.[1]
- Our scheme has low *node congestion*; that is, no node is unfairly burdened with requests from the rest of the network. In our scheme, if every node generates at most t requests, the expected node-congestion is $O(t \log D)$. For comparison, note that, by averaging, the expected node congestion in any scheme must be at least t times the expected number of hops.[1]

[1] This property may possibly be true also in existing schemes, but proving it would require an indirect and more complicated analysis.

- The scheme lends itself to large local area networks, where the local growth rate is large, but many cheap hops are acceptable in practice; see Section 6.7.

In addition, because our system uses a form of prefix routing, the fault tolerance results of [HK03] apply naturally. In practice, one may also use a "hybrid" system that combines our scheme with a ring [RD01a, GGG+03, RGRK, RGRK03].

Organization: The low stretch scheme is based on a construction of an overlay network with an $O(\text{diameter})$ routing data structure. To gain intuition, we outline the construction, focusing on a simplified case, in Section 6.3.2, and defer some details of the general case to Section 6.4. We then use a technique of LAND [AMD04] to extend the routing algorithm to a low stretch object location in Section 6.5. Next, we use a technique of [HKRZ02, HKR03, HKMR04] to adapt the system to arriving and departing nodes in Section 6.6. Finally, we touch upon a heuristic improvement for large local area networks in Section 6.7.

6.3 Algorithm Outline

Our object location scheme is based on an $O(\text{diameter})$ routing overlay that does not rely on a global bound C, as summarized in the following theorem.

Theorem 8. *There exists a randomized routing scheme, such that the expected number of neighbors of each node v is at most $f(\max_{v,r} \rho_{v,r}, \delta_s, \delta_d) \cdot \log^2 D_N$, for some function f, and any message travels a total distance of $O(D)$ en route to its destination ID.*

All previous low stretch object location schemes [PRR97, ZHR+03, AMD04] (and their extensions) use prefix routing, i.e., every hop "corrects" one additional digit by matching it to the destination ID, and hence the ith hop in the routing is guaranteed to have ID that matches the destination ID in the first i digits. In all these systems, the digit size is determined in advance so as to accommodate for the growth rate of the network. In contrast, our scheme does not have a fixed digit size. Instead, we match a varying number of bits in each step, which is essentially like having a variable digit size. This leads to using, in every locality of the network, the optimal digit size for that locale. More precisely, the number of bits matched at a given step depends on the local growth rate $\rho_{v,r}$. As a result, it can vary both over the nodes and over the scales. For example, one message may match two bits at the first hop (scale) and five at the second, while another may match four at the first hop and three at the second; in fact, the total number of bits fixed in the first two hops need not be the same, unless the two messages have the same destination ID and they meet after two hops. See Figure 6.2 for illustration.

The rest of this section outlines the proof of Theorem 8, by focusing on a simplified case. Details of the general construction are given in Section 6.4. We then use this routing overlay in Section 6.5 to get $1 + \epsilon$ stretch by additional "publish" pointers on nodes that are in the vicinity of the search path, similar to LAND [AMD04]. The additional pointers increase storage by a factor that depends on the growth rate and on ϵ. Throughout, we shall assume $\rho_{v,r}$ is known exactly, although it can be easily seen that it suffices to know $\rho_{v,r}$ within a constant factor.

6.3.1 A Simple Scheme

Each node v in the overlay network is assigned a randomly chosen identifier, called an ID. These IDs are sufficiently long bit strings so that no two are the same. Suppose a node u wishes to send a message, given only the destination's ID. If, say, $d_{\min}$ and D are powers of 2, then the message will route through at most $\log D_N$ hops, where the ith hop is, by definition, no longer than $2^i d_{\min}$, as depicted in Figure 6.2(left). Thus, the total length of a routing path is at most $\sum_{i=0}^{\log D_N} 2^i d_{\min} \leq 2D$, i.e., twice the network diameter.

The routing is a variant of prefix routing, where instead of fixing one bit at a time, the number of fixed bits depends on the local growth rate at the corresponding scale. The last hop will fix all the bits, thereby guaranteeing arrival to destination.

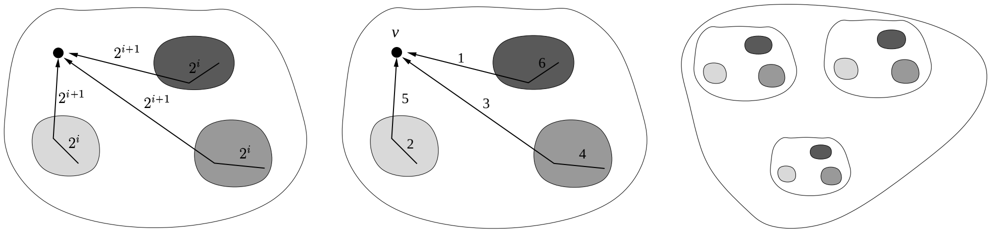

Figure 6.2: *Varying the number of bits at each step.*The leftmost picture depicts the ith and $i+1$st hops in the route of three messages, showing that when $d_{\min} = 1$, each ith hop travels distance at most 2^i. The center picture shows the number of (additional) bits that are fixed along a single hop in these routes. Darker shading represents a higher density of nodes, which leads to fixing more bits, but by the time the messages reach v, all routes fix a total of seven bits. The rightmost picture shows that the left two pictures are only a piece in a larger scheme.

This is in contrast to previous systems, such as PRR [PRR97], Tapestry [ZHR$^+$03], Pastry [RD01a], and LAND [AMD04], which choose a predetermined number of bits to fix at each step; their aim is to get the ith hop travel at most a distance of $2^i d_{\min}$, but this property is only indirectly guaranteed via the growth rate bound. Figure 6.2(center) depicts our bit fixing along two routing hops. Notice that in the dense region (represented by darker shading), more bits are fixed at the earlier hop, but after two hops the same number of bits have been fixed.

Routing Entities: We divide the routing state of a node into *routing entities*, and describe the routing as a transaction between entities. A routing entity corresponds to a level of the neighbor table in [PRR97, ZHR$^+$03, RD01a, AMD04]. A *scale* is a number $2^i \in [d_{\min}, 2D]$, where i is an integer. Each node hosts a *seed entity* for every scale 2^i, and may host additional *emulated* entities, as will be described below.

Each entity has an ID, denoted $E.\mathsf{id}$. The ID of a seed entity is the ID of the hosting node. To determine which messages E may accept, each entity E also has a *prefix requirement*, denoted $E.\mathsf{req}$, that we shall define later and a scale, denoted $E.\mathsf{scale}$. The neighbors of an entity E are all seed entities E' such that[2]

(a). $E'.\mathsf{scale} = 2 \cdot E.\mathsf{scale}$;

[2] Note that this asymmetric definition of being a neighbor provides a directed link from E to E'. The actual implementation of this may be bi-directional.

(b). E' is within distance $E.\mathsf{scale}$ of E; and

(c). $E'.\mathsf{id}$ agrees with $E.\mathsf{id}$ on the $E.\mathsf{req}$ first bits.

Routing Algorithm: A given message passes through entities of increasing scales. Denoting the ith entity in the route as E_i, the first entity E_0 is the minimum scale (i.e. scale $2^i \in [d_{\min}, 2d_{\min}))$ seed entity of the node that generated the message. At the ith step, the entity E_i forwards the message to the entity E_{i+1} that is chosen as the entity E' whose ID prefix matches the destination ID in the most bits possible among all nodes meeting requirements (a), (b), and

(c$'$) $E'.\mathsf{id}$ agrees with the destination ID on the $E'.\mathsf{req}$ first bits.

The routing always terminates when the scale is the message reaches an entity E_i whose scale $E_i.\mathsf{scale}$ is greater than twice the diameter of the network D. Indeed, by definition, the previous entity E_{i-1} stores *all nodes in the network* (more precisely, seed entities of scale at least D) that agree with the destination ID on $E_{i-1}.\mathsf{req}$ bits, so the destination node must be in this list. Notice that the shorter $E.\mathsf{req}$, the more neighbors E will have to store. But if $E.\mathsf{req}$ is long, it can only accept routing requests for objects with a long ID match, which means that other nodes cannot use E as a neighbor quite as easily. Lemmas 2.1 and 2.2 show this trade off.

Unfortunately, there is no guarantee for E_i that an entity matching (a),(b) and (c$'$) exists. For example, suppose an entity E_i with prefix requirement 11 is trying to route a message to ID 11011. If the nodes with prefix 110 are all more than 2^i away, E cannot use any of them in its neighbor table without violating the invariant that the ith hop does not travel too far away. In this case, where no suitable seed entity is found, we use a technique of LAND [AMD04] and let the node create an *emulated* entity with the required ID prefix (e.g., 110) and scale (e.g., 2^{i+1}). This emulated entity stores all the information that a seed entity would.

Note that emulating entities is not quite the same as giving the node v another ID, as that would be the equivalent of creating an entire new node, while an emulated entity only corresponds to a single level of the routing table. Nonetheless, it is important the total number of emulated entities per node is small, and proving this is the main technical difficulty. A standard argument regarding branching processes shows that if the expected number of emulated entities that are *directly* generated by any single entity is upper bounded by a constant $\lambda < 1$ then the total number of emulated entities per seed entity is expected to be a constant. The intuition is that if that expectation was $\lambda \geq 1$, then each seed entity generates, in expectation, λ emulated entities, each of which generates λ more emulated entities, and so on, resulting in an exponential blow up in the total number of neighbors that poor node has to maintain.

The Prefix Requirement: The challenge in defining the prefix requirement is to create a self-organizing overlay network that adapts to the local growth rate, but still ensures that there is a routing path from any locale to any possible destination. A fast growing $E.\mathsf{req}$ (as a function of the scale) means that there will be few matching nodes inside the ball of radius $E.\mathsf{scale}$, and so many emulations will be required. On the other hand, a slow growing $E.\mathsf{req}$ means that there will be many matching nodes, and so E's neighbor table will be large. The crux is to show that one can define a prefix requirement that simultaneously satisfies these two competing needs.

6.3.2 Sketch of Analysis

Before considering the general case (in Section 6.4), we analyze the case where the local growth rate is "smooth" in the sense that $\rho_{v,r} \approx \rho_{w,r}$ whenever v and w are close, i.e., within distance r of each other. Notice that this does not preclude the growth rate from varying considerably between distant points. In this simplified case, we shall scale all the distances in the network so that the smallest distance is 1, and the largest (i.e., the diameter of the network) is D. Hence, a *scale* is a number 2^i for an integer $0 \leq i \leq \lceil \log D \rceil + 1$. we further assume here that all quantities are integral to avoid rounding notation, and denote by log a base 2 logarithm.

For each scale s entity E located at a node v, we set the prefix requirement $E.\mathsf{req}$ to be

$$\mathsf{pref}(v,s) := \log|\mathcal{B}_v(s/2)| - a,$$

where a represents a universal comfort factor.[3] Setting a picks the trade off point between table size and the number of emulated entities. Note that $E.\mathsf{req}$ depends on the node's locale, unlike the analogous prefix requirement in [PRR97, ZHR+03, RD01a, AMD04]. We next show in Lemmas 18 and 19 that our prefix requirement indeed satisfies the two competing needs mentioned above.

Lemma 18. *The expected table size of an entity of scale s on node v is $2^a\rho_{v,s/2}$.*

Proof. There are $|\mathcal{B}_v(s)|$ nodes in the ball of radius s around v. Each one matches v's required prefix with probability $2^{-\mathsf{pref}(v,s)} = \frac{2^a}{|\mathcal{B}_v(s/2)|}$, so the entity's expected table size is $|\mathcal{B}_v(s)| \cdot \frac{2^a}{|\mathcal{B}_v(s/2)|} = 2^a\rho_{v,s/2}$. □

Lemma 19. *If a is chosen large enough, the expected number of directly emulated entities is less than 1.*

Proof. Let $\mathsf{pref}(v,s)$ denote the prefix length required by v at scale s.

An entity E at scale s located on node v requires an emulated entity for every extension of $E.\mathsf{req}$ by $\mathsf{pref}(v,2s) - \mathsf{pref}(v,s)$ bits where E knows of no suitable seed entity E' (in terms of ID, scale, and distance from E). Think of the $\mathsf{pref}(v,2s) - \mathsf{pref}(v,s)$ bits as describing a digit to match. Intuitively, we match as many digits as possible so long as we can find a suitable neighbor in the ball around E of radius $E.\mathsf{scale}$.

Think of the $2^{\mathsf{pref}(v,2s)-\mathsf{pref}(v,s)} = \rho_{v,s}$ possible extensions as bins, and of the nodes within distance s from E as balls. The assumption that $\rho_{v,r} \approx \rho_{w,r}$ means that $\mathsf{pref}(w,2s) = \mathsf{pref}(v,2s)$, and so if w as the right prefix and the right scale, its prefix requirement does not prevent it from being a neighbor of v.

[3] In this simplified exposition a would actually have a weak dependence on the local growth rate. This is eliminated in our general scheme, using a more complicated prefix requirement function.

We then wish to bound the expected number of empty bins. By Lemma 18, the expected number of balls is $2^a \rho_{v,s}$, and, clearly, each of them lands in a random bin. Notice that this is just a coupon collecting problem; so if there are N bins and more than $N \ln N$ balls, the expected number of empty bins falls below a constant less than one. Here, we only know the expected number of balls, but the argument is similar.

It follows that the expected number of directly emulated entities is strictly less than some constant $\alpha < 1$ if $2^a \rho_{v,s} \geq \rho_{v,s} \ln \rho_{v,s}$, so a needs be larger than $\log_2 \ln \rho_{v,r}$. □

The above shows how to build a system when the local growth rate $\rho_{v,r}$ is "smooth". The rest of the paper deals with the case where the local growth rate allowed to vary (in a bounded way) across scales and between nearby points in the network. The proof of this more general case follows the same basic outline as the above.

6.4 More General Networks

In this section we prove Theorem 8 by constructing an $O(\text{diameter})$ routing overlay for the general case. The algorithm and terminology in this section is exactly the same as that in Section 6.3. In particular, recall that the routing state of each node was divided into entities.[4] The difference is that here we need to handle a more complicated network model, which may have significant changes in local growth rates, and hence we employ a less naive definition of the prefix requirement of an entity E and a more involved proof. In particular, we set $E.\mathsf{req}$ of an entity at node v at scale r to be

$$\mathsf{pref}(v,r) := \max\{0, \log |\mathcal{B}_v(r/2)| - c\log(\rho_{v,r}) - a\}.$$

(We round down if it is not an integer.) Here and throughout, log denotes a logarithm to base 2; a and c are parameters; we will show in Section 6.4.2 that setting $c \geq 2$ and

[4] A routing entity corresponds to a level of the neighbor table in PRR [PRR97], Tapestry [ZHR$^+$03], or Pastry [RD01a]. Since we later "duplicate" some levels of the neighbor table via emulation, we need a convenient way to talk about a level.

$a \geq \max\{3 + \log_2 \delta_d, 2 + \log_2(c+2) + \log_2 \log_2(\delta_s \delta_d)\}$ are sufficient for our purposes, i.e., that the expected size of the neighbor table of a node is small. But first we bound the total distance traveled by a message before reaching its destination.

6.4.1 Guaranteed Delivery and O(D) Routing

Let D denote the diameter of the network (i.e., the maximum distance between two points), and let $d_{\min}$ denote the minimum distance between two points. Let $D_N = D/d_{\min}$ be the *normalized diameter*. We shall show that messages are delivered in $O(\log D_N)$ steps, and the total distance traveled is only $O(D)$. Before proving the efficiency, we prove that every message is delivered.

Lemma 20. *The routing network described always delivers any given message.*

Proof. Let the path of message m through the network be entities $E_1, E_2, \ldots, E_i$. At every step, the ID of E_i matches the destination ID in $E_i.\mathsf{req}$ bits. This is proved by induction on i. The base case is E_1, where $E_1.\mathsf{req} = 0$. The third routing property (c′) ensures that a message is sent only to an entity E' that matches the destination ID in at least $E'.\mathsf{req}$ bits, so the inductive step holds.

Finally, when $E.\mathsf{scale}$ is larger than the diameter D, the entity E stores a seed entity for *every* node ID (throughout the network) that match in $E.\mathsf{req}$ bits, so the next hop goes directly to the destination. If the message arrives at the destination at some intermediate scale, it remains there. □

Corrolary 3. *Every message is routed through at most* $\log D + O(1)$ *hops, and the total distance it travels is* $O(D)$.

Proof. The first hop travels distance at most $2d_{\min}$, and this upper bound is doubled at every hop. The final hop is of length at most $2D$. The total distance is $\sum_{i=\lceil \log d_{\min} \rceil}^{\lceil \log D \rceil} 2^i = O(D)$. The number of hops is equal to the number of terms of the sum; this is $\lceil \log D \rceil - \lceil \log d_{\min} \rceil + 1 = \log D_N + O(1)$ hops. □

6.4.2 Space Complexity

We turn to bounding the expected number of neighbors of a node. The high level argument, where dependency on $\rho_{v,r}, \delta_s, \delta_d$ is suppressed by assuming they are constant, goes as follows. By definition, each node hosts $O(\log D_N)$ seed entities. Now for some constant $\lambda < 1$, each seed entity is expected to *directly* generate at most λ emulated entities; where each of those is, in turn, expected to directly generate at most λ additional emulated entities, and so forth. By a standard branching process bound, the total number of emulated entities that a single seed entity is expected generate is at most $\frac{1}{1-\lambda} \leq O(1)$. Finally, the expected number of neighbors of each entity (seed or emulated) is $O(1)$, and therefore each node is expected to have a total of $O(\log D_N)$ neighbors.

Technically, this argument is flawed, because it does not account for certain correlations. First, the upper bound on the branching process requires certain events to be independent, which is not true in our case — the events of emulating a scale r and a scale $2r$ entities at the same node u depend on the IDs of nodes in $\mathcal{B}_u(r)$ and $\mathcal{B}_u(2r)$, respectively, and these two balls are clearly non-disjoint. Furthermore, in some extreme cases (e.g., if the node u is in Hawaii) these two balls might be equal, and then any emulated entity at scale r generates (deterministically) an emulated entity at scale $2r$. In particular, this shows that λ might be as large as 1; as a result, the branching process bound might be as large as $O(\log D_N)$, increasing the final bound of the above high level argument to $O(\log^2 D_N)$, similar to Theorem 8. A second correlation is between the total number of emulated entities generated a single seed entity at a node u, and the number of neighbors of each of these emulated entities — these random variables both depend on the IDs of nodes around u.

In this extended abstract, we prove that the bounds mentioned in the above high level description hold for seed entities. Analyzing emulated entities is more complicated because their existence is correlated with the IDs of nearby nodes, and at the end of this section we briefly explain the technique used to analyze the corresponding conditional probabilities and expectation.

Lemma 21. *The expected number of neighbors for a scale s seed entity located at node v is $O(2^a \rho_{v,s}^c \rho_{v,s/2})$.*

Proof. A scale s seed entity E located at a node v has to store all the scale $2s$ seed entities matching $E.\mathsf{req}$ within distance s of v. There are $|B(v,s)| = \rho_{v,s/2}|\mathcal{B}_v(s/2)|$ nodes within distance s of v, and the probability that any single one matches the prefix requirement is $\frac{2^a \rho_{v,s}^c}{|\mathcal{B}_v(s/2)|}$, so the expected number of matching nodes in $\mathcal{B}_v(s)$ is $2^a \rho_{v,s}^c \rho_{v,s/2}$. □

Lemma 22. *If for every entity, the expected number of emulated nodes it directly generates is at most a constant $\lambda < 1$ independently of the other entities, then the expectation of the total number of entities generated from one seed entity (directly or indirectly) is at most $\frac{1}{1-\lambda} \le O(1)$.*

Proof. For the purposes of this proof, we define an ith degree emulated entity recursively as follows. An ith degree emulated entity as an entity that created by a $i-1$st degree entity. Let a seed entity be a 0th degree emulated entity (that is, one that is not emulated at all). Let X_i be a random variable denoting the expected number of degree-i entities stemming from one seed entity. Then $\mathbb{E}[X_{i+1}|X_i] \le X_i \cdot \lambda$, and taking expectation on both sides gives $\mathbb{E}[X_{i+1}] \le \lambda \cdot \mathbb{E}[X_i]$. Since $X_0 = 1$, we have that $\mathbb{E}[X_i] \le \lambda^i$, and thus $\sum_i \mathbb{E}[X_i] \le 1/(1-\lambda)$. □

Lemma 23. *The expected number of entities directly emulated by a seed entity is at most $1/e$, if $c \ge 2$ and $a \ge \max(3 + \log \delta_d, 2 + \log(c+2) + \log\log \delta_s \delta_d)$.*

Proof. Let E be a seed entity of scale r at node v. Consider a message m that may be routed through E. By definition, this message's destination ID must agree with with $E.\mathsf{id}$ on the first $\mathsf{pref}(v,r) = \max\{0, \log_2|\mathcal{B}_v(r/2)| - c\log_2(\rho_{v,r}) - a\}$ bits. Now fix a node $w \in \mathcal{B}_v(r)$ and let's see whether E can forward this message m to w's scale $2r$ seed entity $w.\mathsf{E}[2r]$. This is possible if the destination ID matches $w.\mathsf{id}$ on the first $\mathsf{pref}(w, 2r)$ bits. Since w's ID is random, this event happens with probability $2^{-\mathsf{pref}(w,2r)}$. Hence,

$$\Pr[E \text{ has to emulate to route } m] \leq \prod_{w \in \mathcal{B}_v(r)} (1 - 2^{-\mathsf{pref}(w,2r)}).$$

Letting $\hat{w}$ be the node $w \in \mathcal{B}_v(r)$ whose prefix requirement $\mathsf{pref}(w, 2r)$ is maximal, we get the upper bound

$$\Pr[E \text{ has to emulate to route } m] \leq (1 - 2^{-\mathsf{pref}(\hat{w},2r)})^{|\mathcal{B}_v(r)|}.$$

Let Y_E be a random variable representing the number of entities directly emulated by E. In order to upper bound $\mathbb{E}[Y_E]$, we need to consider all possible messages m that may be routed through E. The point is that the many possible destination IDs can be grouped into relatively few distinct events. Notice that our arguments above for the message m actually depend only on the first $\mathsf{pref}(\hat{w}, 2r)$ bits of the destination ID, while the first $\mathsf{pref}(v, r)$ bits of the destination ID must be the same as those of $v.\mathsf{id}$. Thus, we can group the possible destination IDs by their contents in positions $\mathsf{pref}(v, r) + 1, \ldots, \mathsf{pref}(\hat{w}, 2r)$. By the analysis above, the probability that (the destination IDs of) a single group requires an emulated entity is at most $(1 - 2^{-\mathsf{pref}(\hat{w},2r)})^{|\mathcal{B}_v(r)|}$.

There are at most $2^{\mathsf{pref}(\hat{w},2r) - \mathsf{pref}(v,r)}$ groups, and hence

$$\begin{aligned} \mathbb{E}[Y_E] &\leq 2^{\mathsf{pref}(\hat{w},2r)-\mathsf{pref}(v,r)} \cdot (1 - 2^{-\mathsf{pref}(\hat{w},2r)})^{|\mathcal{B}_v(r)|} \\ &\leq e^{\mathsf{pref}(\hat{w},2r)-\mathsf{pref}(v,r)-|\mathcal{B}_v(r)|/2^{\mathsf{pref}(\hat{w},2r)}}. \end{aligned}$$

It remains to upper bound the right hand side by $1/e$. This requires some technical calculations, whose intuition is the following. Suppose that any ball of radius r in the network contains about r^d nodes, for all r. If we were to define $\mathsf{pref}(v, r) = \log |\mathcal{B}_v(r)|$, then $\mathsf{pref}(\hat{w}, 2r) - \mathsf{pref}(v, r) \simeq \log[(2r)^d/r^d] = d$, and $|\mathcal{B}_v(r)|/2^{\mathsf{pref}(\hat{w},2r)} \simeq |\mathcal{B}_v(r)|/|\mathcal{B}_{\hat{w}}(2r)| \simeq 1/2^d$, which yields a poor upper bound $\mathbb{E}[Y_E] \leq 2^d \exp\{-2^{-d}\}$. But we can easily overcome this by setting $\mathsf{pref}(v, r) = \log |\mathcal{B}_v(r)| - 3d$, and then Y_E is at most $2^d \cdot \exp\{-2^{-d+3d}\} = o(1)$.

Now, to the actual calculations. Notice that $\mathcal{B}_{\hat{w}}(r) \subseteq \mathcal{B}_v(2r)$, and that $|\mathcal{B}_v(2r)| = \rho_{v,r}\rho_{v,r/2}|\mathcal{B}_v(r/2)|$. Hence,

$$\begin{aligned} \mathsf{pref}(\hat{w}, 2r) - \mathsf{pref}(v, r) &\leq \log \frac{|\mathcal{B}_{\hat{w}}(r)|}{|\mathcal{B}_v(r/2)|} + c \log \frac{\rho_{v,r}}{\rho_{\hat{w},2r}} + 1 \\ &\leq \log(\rho_{v,r}\rho_{v,r/2}) + c \log \frac{\rho_{v,r}}{\rho_{\hat{w},2r}} + 1 \end{aligned}$$

Second, by the definition of the prefix requirement we have

$$\begin{aligned} |\mathcal{B}_v(r)|/2^{\mathsf{pref}(\hat{w},2r)} &= |\mathcal{B}_v(r)| \cdot \frac{2^{a-1}(\rho_{\hat{w},2r})^c}{|\mathcal{B}_{\hat{w}}(r)|} \\ &\geq \frac{2^{a-1}(\rho_{\hat{w},2r})^c}{\rho_{v,r}} \end{aligned}$$

Noticing that $\rho_{v,r/2} \leq \delta_s \rho_{v,r}$ and $\rho_{v,r} \leq \delta_d \rho_{\hat{w},r} \leq \delta_d \delta_s \rho_{\hat{w},2r}$, we obtain

$$\begin{aligned} \mathbb{E}[Y_E] &\leq \exp\{\log(\delta_s(\delta_d\delta_s\rho_{\hat{w},2r})^2) + c\log(\delta_d\delta_s) + 1 - 2^{a-1}(\rho_{\hat{w},2r})^{c-1}/(\delta_d\delta_s)\} \\ &\leq \exp\{2\log\rho_{\hat{w},2r} + (c+3)\log(\delta_d\delta_s) + 1 - 2^{a-1}(\rho_{\hat{w},2r})^{c-1}/(\delta_d\delta_s)\} \end{aligned}$$

To conclude that the right hand side is at most $1/e$, it suffices to show that

$$\begin{aligned} 2\log\rho_{\hat{w},2r} - 2^{a-3}(\rho_{\hat{w},2r})^{c-1}/(\delta_d\delta_s) &\leq 0 \\ (c+3)\log(\delta_d\delta_s) - 2^{a-3}(\rho_{\hat{w},2r})^{c-1}/(\delta_d\delta_s) &\leq 0 \\ 1 - 2^{a-2}(\rho_{\hat{w},2r})^{c-1}/(\delta_d\delta_s) &\leq -1 \end{aligned}$$

Rearranging these three inequalities we get

$$\begin{aligned} \delta_d\delta_s\log\rho_{\hat{w},2r} &\leq 2^{a-4}(\rho_{\hat{w},2r})^{c-1} \\ (c+3)(\delta_d\delta_s)\log(\delta_d\delta_s) &\leq 2^{a-3}(\rho_{\hat{w},2r})^{c-1} \\ 2\delta_d\delta_s &\leq 2^{a-2}(\rho_{\hat{w},2r})^{c-1} \end{aligned}$$

Since $\rho_{\hat{w},2r} \geq 1$, the third inequality holds whenever $a \geq 3 + \log_2(\delta_d\delta_s)$. Since $\log\rho_{\hat{w},2r} \leq \rho_{\hat{w},2r}$, the first inequality holds whenever $c \geq 2$ and $a \geq 4 + \log_2(\delta_d\delta_s)$. Since $\rho_{\hat{w},2r} \geq 1$, the second inequality holds whenever $a \geq 3 + \log(c+3) + \log(\delta_d\delta_s) + \log\log(\delta_d\delta_s)$.

To conclude, it suffices that $c \geq 2$ and $a \geq 3 + \log(c+3) + \log(\delta_d\delta_s) + \log\log(\delta_d\delta_s)$.

□

Analyzing emulated entities: Emulated entities located at the same node are correlated, and therefore the branching process bound cannot be applied to the total number of emulated entities that a seed entity generates. We overcome this by considering a branching process over a subset of the scales, which we call *marked scales.* These marked scales are defined only for the sake of analysis – the algorithm does not change. Starting with the scale of the seed entity, which is always marked, iteratively increase the current scale r by a factor of 2. If $|\mathcal{B}_v(r)|$ is at least twice as large as the ball at the last marked scale, we mark scale r. Let us denote the subset of marked scales by $\{m_i\}_i$; thus, $\frac{|\mathcal{B}_v(m_i)|}{|\mathcal{B}_v(m_{i-1})|} \geq 2$, and $\frac{|\mathcal{B}_v(m_i/2)|}{|\mathcal{B}_v(m_{i-1})|} < 2$. Now a slight generalization of Lemma 23 can bound the number of emulated entities at a scale $r \in [m_i, m_{i+1})$ conditioned on the number of emulated entities in the marked scale m_{i-1} (or the seed entity, if $i = 0$), and this is enough to prove an analogue to the branching process. The main points in changing Lemma 23 are bounding the term corresponding to $\mathsf{pref}(\hat{w}, 2r) - \mathsf{pref}(v, r)$, and arguing that the IDs of at least half the nodes in the ball corresponding to $B(v, r)$ are independent of the data we conditioned on.

Another issue is the expected number of neighbors of an emulated entity. Fixing two seed entities, E at node u and E' at node u', we upper bound the expectation of the total number of links that all the emulated entities generated (directly or indirectly) by E have to the entity E'. The main point is that the expected number of the former entities (which are all of scale $E'.\mathsf{scale}/2$) can be bounded, even if we condition on a fixed ID for u' (because for Lemma 23 it suffices that all nodes but one have random IDs), and now the randomness in the ID of E' makes the number of entities that link to E' have small expectation.

6.4.3 Load balance

We now consider the load balance of our scheme, measured as the number of message routed through any single node, when every network node is the source of at most one message to a random ID. The bound scales linearly by linearity of expectation, if every node is the source of at most t messages.

Lemma 24. *Consider a routing where every node is the source of at most one message with a fixed destination ID. The total number of messages expected to route through any single node w (over the random choices of node IDs) is at most $O(2^a \rho_{w,r}^{c+1} \rho_{w,r/2} \log D)$.*

Proof. Fix a node $w \in X$, and consider a message sent from source node u toward a given destination ID. The number of hops in our routing scheme is $O(\log D)$ by Lemma 3. Let v_r be the node visited in the scale r hop along the route. We know that $d(v_{r/2}, v_r) \leq r$, and thus $d(u, v_r) \leq r + r/2 + \cdots < 2r$. Thus, w can only be the ith hop in the route if $u \in \mathcal{B}_w(2r)$. Furthermore, for w to be the scale r hop in the route it is necessary that it matches the destination ID on a prefix of length $\mathsf{pref}(w, r)$. We shall consider only hops visiting a seed entity at w, because if the route can contain an emulated entity then it must also contain at least one seed entity at the same node. Now for a seed entity, since w's ID is random, the prefix match happens with probability $1/2^{\mathsf{pref}(w,r)}$. Therefore, the expected number of source nodes (and thus messages) that use w as their ith hop is at most

$$|\mathcal{B}_w(2r)| \cdot \frac{1}{2^{\mathsf{pref}(w,r)}} \leq \frac{|\mathcal{B}_w(2r)| \cdot 2^a \rho_{w,r}^c}{|\mathcal{B}_w(r/2)|} = 2^a \rho_{w,r}^{c+1} \rho_{w,r/2}.$$

□

6.5 Object Location

This section shows how to use the overlay network construction described in the previous section to do low-stretch object location. The routing algorithm and its geometrically increasing hop lengths will play a key role in achieving low stretch. We use the notation of Section 6.4.

Objects are placed in the network by their *publisher* node. An object location data structure (DOLR) such as the one presented here determines where to place pointers to the objects so that a *searcher* node is able to locate object copies efficiently. We measure the efficiency of an object-location request in terms of *stretch*, the ratio between the total distance traveled searching for the object and the distance to the closest copy of the object. It is possible to achieve a stretch of one by placing pointers to the object at all potential searchers; this is too much. In this section, we show how to place a small number of pointers and get $O(1)$ stretch. By placing more pointers in a manner similar to LAND [AMD04], $1+\epsilon$ stretch is possible.

The routing algorithm forms the basis for the object location algorithm. The name of the object is hashed into the same name space as the node IDs, and the publisher of the object routes toward the object's ID. The node with the ID most closely matching the object's ID is the *root*. The publisher then places pointers along the routing path from the object's location to object's ID (i.e., the root). To search for the object, the searcher routes toward the object's ID. Suppose that a searcher and a publisher are within distance r of each other. If they happen to have the same scale-r routing hop when routing toward the object's ID, then the searcher finds a pointer to the object at that hop and is able to shortcut the rest of the routing. In the worst case, however, the first hop they share might be of a scale much larger than r (such as the network's diameter).

To solve this, the publisher places pointers to the object at any possible scale-r routing entity that a searcher within distance r might visit when looking for the object. This yields $O(1)$ stretch, as we describe below. We call this new set of neighbors "publish neighbors," and the old set of neighbors are "routing neighbors".

More precisely, for an entity E at scale $E.\mathsf{scale}$ with prefix requirement $E.\mathsf{req}$, maintain links to all scale $E.\mathsf{scale}$ seed *and emulated* entities E' within distance $5 \cdot E.\mathsf{scale}$ of E matching in $\min\{E'.\mathsf{req}, E.\mathsf{req}\}$ bits. Objects are published by routing toward the object's root. At every hop, pointers to the object are placed on the publish neighbors of the current entity. Search routes toward the object's root, checking for

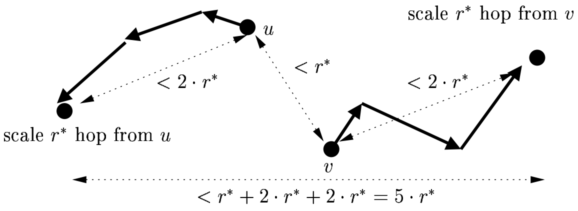

Figure 6.3: *Triangle Inequality.*The routes of messages that have the same destination ID. If the two source nodes are within distance r^*, their corresponding scale r^* hops are within $5r^*$ of each other.

a pointer to the object at each entity en route.[5] We now prove that this results in $O(1)$ stretch.

Lemma 25. *If node y searches for an item published by node x, the total length of the search path before a pointer to the object is found is at most $18d(x, y)$.*

Proof. Let x_r be the scale r hop on the path toward the root from x, and let y_r be the scale r hop on the path toward the root from y. Choose r^* such that $r^*/2 \leq d(x, y) \leq r^*$. The key part of the proof is to note that $d(x_{r^*}, y_{r^*}) \leq 5r^*$. Figure 6.3 shows why this is so. In particular, notice that $d(x, x_{r^*}) \leq 2r^*$, and $d(y, y_{r^*}) \leq 2r^*$, then $d(x_{r^*}, y_{r^*}) \leq d(x, y) + d(x, x_{r^*}) + d(y, y_{r^*}) \leq 5r^*$. When x_{r^*} received the publish request for the item, it sent an object pointer to y_{r^*} (since y_{r^*} is within distance $5r^*$ and matches in the right number of bits). Thus, when y searches for an object, it has to pay at most the distance to y_{r^*} and then the distance from y_{r^*} to the object. The first distance is bounded by $2r^*$, and the second distance is at most $d(y_{r^*}, x_{r^*}) + d(x_{r^*}, x)$. Hence, the overall the length is at most $2r^* + 5r^* + 2r^* \leq 9r^*$, and since $d(x, y) \geq r^*/2$, the stretch is at most 18. □

It remains to bound the expected number of publish neighbors a node has to maintain. Similar to the discussion in Section 6.4, we analyze below only the case

[5] An alternative would be to keep the publishing only along the path to the root and have the searcher look in many places at every scale. (See Awerbuch and Peleg[AP91] and their discussion of read and write sets.)

of seed entities; the analysis can then be extended to emulated entities using similar techniques to those sketched in Section 6.4.

Lemma 26. *The expected number of publish neighbors for a scale s seed entity at node v is $O(2^a\delta_d^{5+c}\rho_{v,8s}\rho_{v,s}^{c-1}\delta_s^{10+3c})$.*

Proof. A scale s entity E located at a node v has to store all the nodes matching $E.\mathsf{req}$ within distance $5s$ of v. There are $|B(v,s)| = \rho_{v,s/2}|\mathcal{B}_v(s/2)|$ nodes within distance s of v, so there are $\rho_{v,s/2}\rho_{v,s}\rho_{v,2s}\rho_{v,4s}|\mathcal{B}_v(c)|$ within $8s$ of v, which bounds the number within $5s$ of v. The probability that any single one matches the prefix requirement of E is $2^{-\mathsf{pref}(v,s)} = \frac{2^a\rho_{v,s}^c}{|\mathcal{B}_v(s/2)|}$, so the expected number of matching nodes in $\mathcal{B}_v(8s)$ is $2^a\rho_{v,4s}\rho_{v,2s}\rho_{v,s}^c\rho_{v,s/2}$.

However, this only counts nodes that match v's prefix requirement. But recall that v needs to connect to all entities u such that match in $\min\{\mathsf{pref}(u,s),\mathsf{pref}(v,s)\}$, so the number of entities actually stored by v can be a factor of $2^{\mathsf{pref}(v,s)-\mathsf{pref}(u,x)}$ greater.

Consider some entity E located at a node u such that $d(u,v) \leq 8s$. Since $|B(u,16s)| \geq |B(v,8s)|$, and using the definition of δ_s and δ_d, we get

$$\begin{aligned}
&\mathsf{pref}(v,s) - \mathsf{pref}(u,s) \leq \\
&\quad\leq \log\frac{|B(u,16s)|}{|B(v,8s)|} + \log\frac{|B(v,s/2)|}{|B(u,s/2)|} - c\log\frac{\rho_{v,s}}{\rho_{u,s}} \\
&\quad\leq \log\frac{\rho_{u,8s}\rho_{u,4s}\rho_{u,2s}\rho_{u,s}\rho_{u,s/2}}{\rho_{v,4s}\rho_{v,2s}\rho_{v,s}\rho_{v,s/2}} + c\log\rho_{u,s} \\
&\quad\leq \log\frac{(\delta_d\rho_{v,8s})^5\delta_s^{10}}{\rho_{v,4s}\rho_{v,2s}\rho_{v,s}\rho_{v,s/2}} + c\log\delta_s^3\delta_d\rho_{v,8s} \\
&\quad\leq \log\frac{\delta_d^{5+c}\rho_{v,8s}^{5+c}\delta_s^{10+3c}}{\rho_{v,4s}\rho_{v,2s}\rho_{v,s}\rho_{v,s/2}}.
\end{aligned}$$

Thus, v may need to match a factor of $\frac{\delta_d^{5+c}\rho_{v,8s}^{5+c}\delta_s^{10+3c}}{\rho_{v,4s}\rho_{v,2s}\rho_{v,s}\rho_{v,s/2}}$ more nodes than the $2^a\rho_{v,4s}\rho_{v,2s}\rho_{v,s}^c\rho_{v,s/2}$ that match its prefix requirement, which is at most $O(2^a\delta_d^{5+c}\rho_{v,8s}\rho_{v,s}^{c-1}\delta_s^{10+3c})$.

This counts the number of seed entities stored. But at a given scale, in expectation, each seed entity hosts only a constant number of emulated entities, so the bound still holds even when emulated entities are considered. □

Counting the number of publish neighbors from an emulated entity is more complicated, but can be done using the ideas mentioned in Section 6.4.

Finally, notice that we can reduce the number of publish neighbors if we make the search algorithm more extensive. If nodes both push pointers to their publish neighbors *and* search for objects on their publish neighbors, it is sufficient for each node to keep only those entities in $\mathcal{B}_v(5s)$ that match in $\mathsf{pref}(v, s)$ bits, resulting in a total of $O(2^a \rho_{v,4s} \rho_{v,2s} \rho_{v,s}^c \rho_{v,s/2})$ publish neighbors for a seed entity. Using the notation of Lemma 25, suppose that a search request from y found a publish pointer at y_{r^*} search in the old scheme. Then in the new scheme, either it finds a pointer at y_{r^*} as before, or y_{r^*} searches for the item on x_{r^*}.

6.6 Dynamic System

In order to be practical, this system must be able to adapt to arriving and departing nodes. To build a table for a node v and scale r, we need

1. The number of nodes within distance r of v.
2. The local growth rate around v at scale r (i.e., $\rho_{v,r}$).
3. All the entities matching $\mathsf{pref}(v, r)$ within r of v.

Items (1) and (2) are needed to determine $\mathsf{pref}(v, r)$. Notice that they are equivalent; given (1) for all r, one can compute (2), and vice-versa. While the problem of computing (1) and (2) is new, other systems [HKRZ02, ZHR$^+$03, RD01a, AMD04] have considered (3).

A first approach to estimating (1) and (2) is to find a physically nearby node, and copy the values from there. Intuitively, this should work well, though it may be hard to prove performance results, especially as the network evolves over time. Section 6.6.2 gives a more rigorous technique.

Likewise, finding approximate neighbor lists can also be done easily, if, as is typical in these systems, the data structures need only be close to the correct ones. In this case, the algorithms of [RD01a, ZHR$^+$03] can be used. These algorithms start with a physically nearby node as before, copying its scale r_0 (where r_0 is the smallest scale) neighbor table. Using this list, the node routes toward its own ID, and takes that node's scale $2r_0$ neighbors as its $2r_0$ scale neighbors, and so on. This would be complicated by the different prefix requirements, but some immediate heuristics to deal with this suggest themselves. These lists can be optimized by asking all the scale r neighbors for their neighbors measuring the distances to these neighbors of neighbors, and updating the neighbor table appropriately.

The rest of this section sketches provable techniques for finding (1)-(3). We first show how to find all nodes with a given prefix within distance d of a particular starting point, solving problem (3), and then explain how this primitive can be used for finding the growth rates and the number of nodes in the ball of radius r.

6.6.1 Neighbors

The basic idea is to use the backward-routing techniques first described in [HKRZ02] and later refined in [HKR03, HKMR04]. (Because of the emulated nodes, the algorithm also resembles [KL04b].) We sketch the algorithm here. Consider particular entity E. Its *children* are the entities that have E as a neighbor, and its grandchildren are the entities that have children of E as a neighbor, and so on. Also, if an entity E generates and entity E', we say that E is a child of E'. Define descendants analogously as all entities on the routing path to entity E. The set of descendants form a tree. Note that all scale $d_{\min}$ entities are descendants of every scale D entity.

Lemma 27. *All the descendants of a scale r entity are within distance r of the entity.*

Proof. The proof is by induction. For the base case, consider an entity at scale $d_{\min}$. It has no children, and so all its descendants are within $d_{\min}$. Now consider an entity E at scale r. Its children are all within $r/2$ by construction. If it has a child further than $r/2$ away, that child could not keep E as a neighbor. Further, we know (by the inductive hypothesis) that all the descendants of those child-entities are within $r/2$, and so they are within $r/2 + r/2 = r$ of E. □

To build the table for a node v, we need an algorithm to find all entities within distance d of v. The above suggests the following algorithm. Start with any scale D entity E (recall that D is the largest possible scale). Then, the set S_r will contain all the scale r descendants of E that are ancestors of entities within d of v. Start with $S_D = \{E\}$. Then, given S_{2r}, we find S_r as follows:

1. For each entity e in S_{2r}, put the children of e (all of which have scale r) in S_r.
2. For each entity e in S_r, measure $d(e, v)$ (the distance to v). If this distance is more than $r + d$, then remove e from S_r.

Lemma 28. *If u is within distance d of v, the above algorithm never removes an ancestor of u.*

Proof. Denote u's scale r ancestor by E_u^r. By Lemma 27, we know that $d(u, E_u^r) \leq r$. By assumption, $d(u, v) \leq d$, so by the triangle inequality, $d(v, E_u^r) \leq r + d$, so for each r, E_u^r is kept in the set S'. □

The above algorithm finds all nodes within d starting with *any* scale D entity. However, recall that v needs to find all the nodes with a given prefix within d. Following nearly the same algorithm, we can get this for free by carefully selecting the starting entity so that its ID matches the desired ID. Suppose, more precisely, that we want all the nodes within d of v that match a particular ID α in at least k bits. Then S_D is initialized to contain a scale D entity matching α in at least k bits. Second, we add an additional step to the algorithm.

3. For each e in S_r, if e does not match α in at least k bits, remove e from S_r.

Notice that a search for the neighbors at r finds the scale $r' > r$ neighbors with only a little additional work, so by essentially running this algorithm once, a node v can find neighbors for all the entities it hosts. Using techniques from [HKRZ02, HKR03, HKMR04], one can bound the work in terms of the growth rate and the number of neighbors returned.

6.6.2 Finding growth rate

The previous sections sketch an implementation of a primitive that can be used to find all entities with a specified prefix within a given distance efficiently. Because the prefixes are assigned randomly, this function can be used to get a random sample of the nodes at the rate $1/2^i$ for any i by picking a prefix of length i.

This primitive can be used to estimate the number of nodes in $\mathcal{B}_v(r)$ and $\rho_{v,r}$. Pick a prefix of length i. Then to get an estimate of the number of nodes in $\mathcal{B}_v(r)$, count the number of nodes (ignore any non-seed entities) matching in i bit prefix and multiply by 2^i. The error in this technique can be made small if i is chosen so that there are enough nodes matching the prefix in $\mathcal{B}_v(r)$. Estimates of the size of $\mathcal{B}_v(r)$ and $\mathcal{B}_v(2r)$ can be used to calculate the $\rho_{v,r}$.

Other work [GMS04, LS03] has shown how to sample randomly from a peer-to-peer network, but these techniques give a random sample from anywhere in the network, rather than a sample from within a certain radius, so their results do not seem applicable here.

6.7 Trading stretch and storage in certain subnetworks

In a large local area network (LAN), where the local growth rate is large but hops are cheap, it may be acceptable to perform relatively many hops and give up on the low stretch guarantee for nearby objects. This can be easily achieved in our scheme by a simple modification – routing at sufficiently low scales proceeds by fixing only a single bit at every hop. This requires that every node maintains a seed entity for every single bit that may be fixed, but the number of hops in a route would be logarithmic in the number of nodes in the LAN. For example, if a LAN contains 1024 nodes, then instead of fixing 10 bits at the first hop, each of the first 10 hops will fix only on additional bit, so routing to any node in the LAN would take only 10 hops, and every node only maintains 10 neighbors, instead of 1024.

Further, in a LAN, because the distances within in the LAN are small compared to to the distance to the outside of the LAN, the higher scales are unaffected. Proving this in a more general metric becomes difficult, however.

Chapter 7

Conclusion and Future Work

This thesis describes a self-organizing low-stretch object location system for peer-to-peer networks. The scheme we presented met the three requirements for peer-to-peer networks: it is self-organizing, complete, and load-balanced. Furthermore, in the class of growth-restricted spaces, it has low stretch. In Chapter 6, we present a scheme that gives constant stretch in a wider class of metric spaces, and tied the work in growth-restricted spaces to results for related problems in general metric spaces.

The biggest take-away point is that the network matters. Focusing solely on optimizing a DHT, it is easy to forget that there is an underlying network structure. Yet underlying topology does matter. It affects performance of algorithms for neighbor search, such as PNS(k).

It also affects object location. Because Tapestry and Pastry are so closely related, with Tapestry adding intermediate object pointers, it is tempting to consider object location data as an optimization for DHTs. But doing this gives no sense of about where to place object pointers, or even whether placing object pointers will help at all. The answer to those questions depends on the underlying network. This connection is implicit in the original PRR-tree construction, which only has provably bounds when the digit size is a function of the growth rate of the network. But we draw out that connection more clearly in Chapter 6, where we show a system that picks a digit size that depends on the network.

The remainder of this chapter discusses related issues, leading to several open questions.

Peer-to-peer workloads:

In the introduction, we argued that low stretch is important, and this thesis shows that it is a reasonable goal.

The value of low stretch depends on the workload. If an object can be cached locally after the first access, it may be acceptable to pay a high cost for the first access. Even when objects can be cached after the first access, reducing the cost of the first access might be important. Consider, for example, mp3 files. Typically, people download those once only, but the traffic spent downloading is still significant as shown by Gummadi et al. [GDS$^+$03].

In some cases, it may not be possible to cache objects locally. If the end devices are small, like cell phones, they do not have enough storage for many objects. Also, consider the case of dynamic objects, like a room reservation list. It is critical to always have the updated copy, (to avoid two people scheduling a room at the same time). Keeping local copies on the desktop of every potential users means updating those copies any time a room is reserved.

Another consideration is the quality of the object placement schemes. If data accesses occur from essentially random locations, then even a ideal replica placement scheme may not be able to place objects such that low stretch is useful.

Question 1. *For typical peer-to-peer workloads, is the cost of including intermediate pointers (in terms of network traffic, storage, and complexity) worth the benefits of low stretch?*

Low Stretch in Real Networks: This work has focused on somewhat simplistic models of the network. One question is evaluate the techniques presented here in a more realistic model of a peer-to-peer network.

The algorithms we presented focused on maintaining the completeness property, and ignoring the intermediate object pointers. If the intermediate pointers never get fixed, they are are useless, yet for low stretch, these pointers must be maintained.

Most of the results presented here apply to growth-restricted spaces. But real networks are not growth restricted, so the question is whether these techniques apply. Simulation results suggests that these techniques are at least useful heuristics, but quantifying the benefit of the intermediate pointers in a real network has not yet been done, in part because determining the right workload model is difficult.

Question 2. *Can intermediate object pointers for object location be efficiently maintained in networks with high node turnover? Are realistic networks structured such that placing a few object pointers gives a significant decrease in stretch?*

Trade offs between stretch, load-balance, and the network:

The changing c result of Chapter 6 suggests that there is a trade off between stretch, load balance, and the metric space. Results from [KL04a] giving lower bounds on approximate nearest neighbor search may be related.

Question 3. *What is the relationship between stretch, load-balance, and the properties of the underlying network?*

Self-organization in general networks: An efficient self-organizing scheme for general metric spaces may be impossible, as the general metric space schemes seem to require finding the closest from some set, and finding the nearest neighbor is a hard problem in general.

Perhaps systems exist which can provide a more adaptable trade off, getting logarithmic stretch and storage in general metrics, while getting constant stretch and storage in growth-restricted networks. We approached this issue in Section 6.7 of Chapter 6, showing that in some cases, it was possible to give up on stretch at some levels of the network without losing all stretch guarantees.

Question 4. *Does there exist a self-organizing object location data structure for general networks provable bounds? If so, in growth-restricted networks, is the cost comparable to current schemes for those networks?*

Congestion: This work has focused on minimizing latency for the end user. But the system designer might be more concerned with the stress on the network. On measure of that is the *congestion*, the maximum ratio of the traffic across a given edge and the bandwidth of the edge. Because delays in the Internet are often caused by excessive congestion, lower congestion means improved performance.

When the traffic demands (*i.e.*, how much peer A sends to peer B) are known in advance and paths are randomized, , the optimal routing is easy to find. However, in many contexts, the demands are not known in advance. Recent work has focused on finding routing paths that work relatively well for *any* set of demands.

The *oblivious ratio* of a routing algorithm is the worst-case ratio over the set of demands of the algorithm's maximum congestion to the minimum congestion for that set of demands. Thus, the oblivious ratio measures the cost of not knowing the demands.

Räcke [Räc02] gave a construction of routes with oblivious ratio $O(\log^3 n)$ using trees, resulting in simple routing. Unfortunately, his construction was not efficient, requiring the solution of NP-hard problems. There do exist polynomial time constructions [**?**, BKR03] of these trees, but they are not self-organizing. (Note that Azar et al. [ACF$^+$03] gave non-tree based solution via linearly programming solution, and Applegate and Cohen [AC03] simplified the linear program.)

Question 5. *Can a low congestion network be built in a distributed (self-organizing) way?*

Appendix A

The boundary problem

In this section, we show in a simple example that demonstrates two important points.

- Low average stretch does not imply "low stretch" in the adversarial sense.
- Even some simple networks seem to require extra object pointers to deal with what Section 1.2.2 called the boundary problem.

We consider an idealized PRR-tree on one of the simplest networks: a line. On this very simple network, we argue that the stretch between adjacent nodes is $O(\log n)$, but $O(1)$ for pairs that are far apart.

Though this is far from a realistic model, it demonstrates how the boundary problem in a clean way and seems to predict (at least qualitative) results of simulations in a more realistic model.

Place n overlay nodes at the integers on the number line, from 1 to n, such that adjacent nodes are separated by a distance of one. (We assume that n is a power of 2.) A particular object ID and the routes taken to that ID through the overlay create a logical tree on these nodes. Specifically, in a base-2 Tapestry tree, roughly one-half of the nodes are in the bottom level in the tree, one-quarter are in the next level, etc. Suppose the tree is "perfect", meaning that exactly every other node is a leaf, and consider the pair of nodes $(2k-1, 2k)$, for some $k > 0$. If $2k$ is a leaf, then $2k$'s distance to its parent $2k-1$ is one, and $2k-1$'s distance to its parent (itself) is zero; the average distance from a node to its parent is $\frac{1}{2}$. To simplify the calculation,

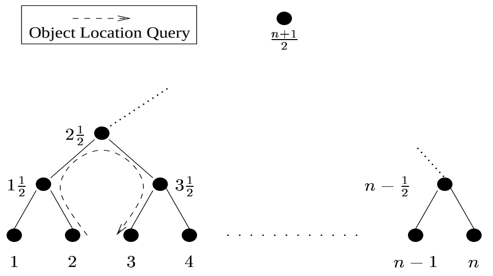

Figure A.1: *A perfect base-2 Tapestry tree, with hypothetical average parents.* A sample object location query with an stretch of three is shown.

imagine a hypothetical average parent located at a distance of $\frac{1}{2}$ from both nodes, at $2k-\frac{1}{2}$. These are not physical nodes; rather, they represent the average length of the first hop of a route. We repeat this process at each level of the logical routing tree (see Figure A.1).

Recall that a *put* message places pointers to them at each node along the path from the publisher to the root, and object location proceeds by checking for pointers along the path from the query source to the root. The stretch is determined by where the publish and search path first intersect, or in other words, the least common ancestor of the publisher and the query source.

Theorem 9. *In this idealized network, the average stretch between adjacent pairs is $O(\log n)$.*

Proof. We consider only the situation when publisher and searcher are at distance one from each other. Half of such pairs of nearby nodes share the same parent, and so have stretch of one, since the request need only travel a distance of one to reach the publisher.

However, because of the boundary problem, one half of the adjacent nodes do not share a parent. Of those, half of them do share a grandparent. For these nodes, the stretch is three (see Figure A.1 for an example).

The argument continues. In general, if $i = \log_2 n$, for any positive $j \leq i$ there are 2^{i-j} pairs with location stretch $2^j - 1$. The average stretch is then $\frac{1}{n}\sum_{j=1}^{i} 2^{i-j}(2^j-1)$, or $i - \sum_{j=1}^{i}\frac{1}{2^j}$, which is less than or equal to $\log n - 1$. Thus, the average stretch is

$O(\log n)$. □

A similar argument shows that for pairs at distance f, the average stretch is $O(\lceil \log(n/f) \rceil)$.[1]

This means that for pairs at least $n/2$ apart, the stretch is $O(1)$. Since most pairs are far apart (one-quarter of all pairs are at distance more than $\frac{n}{2}$), the result is that overall average stretch is low.

Claim 1. *Realistic topologies that have short stretch at long distances may still long stretch between nearby pairs.*

Our thought experiment predicts that since the stretch experienced by a pair of nodes depends on their least common ancestor in the tree, the number of pairs with a least common ancestor at level k to decrease exponentially in k, while the stretch experienced should increase exponentially.

Figure 2.6 bins the stretch between nearby pairs with Tapestry on a simulated transit-stub graph confirms that line example does seem to describe what happens in this topology. This histogram shows that although the location stretch of most queries between nearby nodes is small, for some queries it is very large.

Finally, note that the overall mean stretch for queries between any two nodes is just 3.01 (less than half of the mean stretch for close objects), demonstrating that indeed overall measurements can obscure information about the stretch between nearby nodes.

Discussion: The central problem is the boundary problem. That is, a point k has to associate itself with either the node to the left, (increasing its distance to the right), or to the right (increasing its distance to the left). While there are few nodes on the boundary at at higher levels of the hierarchy, the cost of being on the boundary also increases, so the increase in stretch due to the boundary problem remains roughly the same at each level.

[1] Note that since $f \leq n-1, \lceil \log(n/f) \rceil \geq 1$.

The argument for the line can be extended to networks that are based on d-dimensional grids. In fact, it becomes more serious, since in a d-dimensional grid a point is close to more points, and may have to choose among many more different directions than two.

To reduce stretch, objects could be published to more than one “close” peer, (by having two parents, for example). This is done in the PRR [PRR97] scheme, or LAND [AMD04], and in Chapter 6 of this thesis.

On the line, the cost of doing this is quite small, only doubling the number of pointers. However, in other networks, the cost getting constant stretch this way could be quite high. In networks where all nodes are equidistant from each other, getting constant stretch seems to require placing pointers at nearly all the nodes, concentrated many pointers on a few nodes, or for each node to store a complete list of the other participants. Thus, it seems that the network may forces a trade off between stretch, space, and load-balance.

Natural hierarchical networks: Some networks are naturally hierarchical, and lend themselves to hierarchical search. Consider, for example, a transit-stub network. (See Figure **??**, for example.) Within each stub domain, distances are short, and between stub domains, distances are long.

As long as the regions are the stubs, there is no boundary problem, as each node is only close to one regional directory—that of its own stub. In contrast, even in the line example, there is no placement of directories that does not send adjacent nodes to different directories. Thus, in a transit-stub network, efficient hierarchical search places one directory per stub. (At some level of the hierarchy.)

Finally, note that this is the property identified by Castro et al. [CDHR02] and Gummadi et al. [GGG$^+$03] as *local convergence*. They count the number of exit points of a stub for a given object. Their notion of exit points is the same as our notion of regional directories. Figure A.2 shows the effect of an optimization that attempts to ensure there is only one directory in the stub. Essentially, we guess that if the next

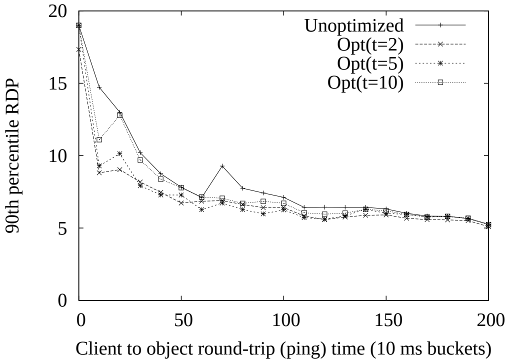

Figure A.2: *The effect of "misrouting".* Misrouting is a technique that ensures that at some level of the hierarchy, there is exactly one directory per stub, if t is guessed correctly.

hop is t longer than the previous hop, the link goes outside the stub, and we route slightly differently. While it does reduce stretch, it does not perform as well as other optimizations (described in Chapter 2), suggesting that even transit stub networks are not natural hierarchies at all levels.

Bibliography

[ABC$^+$02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, December 2002.

[ABNLP89] B. Awerbuch, A. Bar-Noy, N. Linial, and D. Peleg. Compact distributed data structures for adaptive routing. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 479–489, 1989.

[AC03] D. Applegate and E. Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs. In *Proc. of SIGCOMM*, 2003.

[ACF$^+$03] Y. Azar, E. Cohen, A. Fiat, H. Kaplan, and H. Räcke. Optimal oblivious routing in polynomial time. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computi ng*, June 2003.

[ACL03a] Marta Arias, Lenore J. Cowen, and Kofi A. Laing. Compact roundtrip routing with topology-independent node names. In *Proceedings of 22nd Annual Symposium on Principles of Distributed Computing*, pages 43–52, 2003.

[ACL$^+$03b] Marta Arias, Lenore J. Cowen, Kofi A. Laing, Rajmohan Rajaraman, and Orjeta Taka. Compact routing with name independence. In *Proceedings of the 15th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 184–192, 2003.

[AGM$^+$04] Ittai Abraham, Cyril Gavoille, Dahlia Malkhi, Noam Nisan, and Mikkel Thorup. Compact name-independent routing with minimum stretch. In *Proceedings of the 16th Annual Symposium on Parallelism in Algorithms and Architectures*, June 2004.

[AMD04] Ittai Abraham, Dahlia Malkhi, and Oren Dobzinski. LAND: Stretch $(1 + \epsilon)$ locality-aware networks for DHTs. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 543–552, January 2004.

[AP90] Baruch Awerbuch and David Peleg. Sparse partitions (extended abstract). In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 503–153, 1990.

[AP91] Baruch Awerbuch and David Peleg. Concurrent online tracking of mobile users. In *Proc. of SIGCOMM*, pages 221–233, 1991.

[Bar96] Yair Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, 1996.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communication of the ACM*, 18(9):509–517, 1975.

[BKR03] Marcin Bienkowski, Miroslaw Korzeniowski, and Harald Räcke. A practical algorithm for constructing oblivious routing schemes. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 24–33, 2003.

[Bou85] Jean Bourgain. On Lipschitz embedding of finite metric spaces in Hilbert space. *Israel J. Math*, 52:46–52, 1985.

[CCW$^+$] Byung-Gon Chun, Kamalika Chaudhuri, Hoeteck Wee, Marco Barreno, Christos Papadimitriou, and John Kubiatowicz. Selfish caching in distributed systems: A game-theoretic analysis. To appear in PODC 2004.

[CDG$^+$02] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the 5th Symposium on Operating Systems Desgin and Implementation*, pages 299–314, 2002.

[CDHR02] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Anthony Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *Proceedings of the International Workshop on Future Directions in Distributed Computing*, pages 52–55, June 2002.

[CKK02] Y. Chen, R. H. Katz, and J. D. Kubiatowicz. Scan: A dynamic, scalable, and efficient content distribution network. In *Proceedings of the International Conference on Pervasive Computing*, August 2002.

[Cla97] Kenneth L. Clarkson. Nearest neighbor queries in metric spaces. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1997.

[DKK$^+$01] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, October 2001.

[Dou02] John Douceur. The Sybil attack. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, pages 251–260, 2002.

[DW01] John R. Douceur and Roger P. Wattenhofer. Large-scale simulation of replica placement algorithms for a serverless distributed file system. In *Proceedings of MASCOTS*, 2001.

[DZD$^+$03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common API for structured P2P overlays. In *Proceedings of IPTPS*, pages 33–44, 2003.

[Fan] Shawn Fanning. Napster. `http://www.napster.com`.

[FS02] Amos Fiat and Jared Saia. Censorship resistant peer-to-peer content addressable networks. In *Proceedings of Symposium on Discrete Algorithms*, 2002.

[GDS$^+$03] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 19th ACM SOSP*, October 2003.

[GG01] Cyril Gavoille and Marc Gengler. Space-efficiency for routing schemes of stretch factor three. *Journal of Parallel and Distributed Computing*, 61(5):679–687, May 2001.

[GGG$^+$03] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. of SIGCOMM*, pages 381–394, 2003.

[GLR03] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient routing for peer-to-peer overlays. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, pages 113–126, March 2003.

[GMS04] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random walks in peer-to-peer networks. In *Proceedings of IEEE INFOCOM*, 2004.

[GP96] Cyril Gavoille and Stephane Perennes. Memory requirements for routing in distributed networks (extended abstract). In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 125–133, May 1996.

[GSG02] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. In *Proceedings of the second ACM SIGCOMM Workshop on Internet measurment*, pages 5–18, 2002.

[HHB$^+$03] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham Boon, Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with pier. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, September 2003.

[HK03] Kirsten Hildrum and John Kubiatowicz. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *Proceedings of the 17th International Symposium on Distributed Computing*, pages 321–336, 2003.

[HKK04] Kirsten Hildrum, Robert Krauthgamer, and John Kubiatowicz. Object location in realistic networks. In *Proceedings of the 16th Annual Symposium on Parallelism in Algorithms and Architectures*, June 2004.

[HKMR04] Kirsten Hildrum, John Kubiatowicz, Sean Ma, and Satish Rao. A note on finding the nearest neighbor in growth-restricted metrics. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 553–554, 2004.

[HKR03] Kirsten Hildrum, John Kubiatowicz, and Satish Rao. Another way to find the nearest neighbor in growth-restricted metrics. Technical Report UCB/CSD-03-1267, UC Berkeley, 2003.

[HKRZ02] Kirsten Hildrum, John Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. In *Proceedings of the 14th Annual Symposium on Parallel Algorithms and Architectures*, pages 41–52, 2002.

[HKRZ03] Kirsten Hildrum, John Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed object location in a dynamic network. *Theory of Computing Systems*, March 2003.

[KL04a] Robert Krauthgamer and James R. Lee. The black-box complexity of nearest neighbor search. In *31st International Colloquium on Automata, Languages and Programming*, July 2004. To appear.

[KL04b] Robi Krauthgamer and James R. Lee. Navigating nets: Simple algorithms for proximity search. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 791–801, January 2004.

[KLL$^+$97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM Press, 1997.

[KR02] David Karger and Matthias Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 741–750, 2002.

[KR03] Bong-Jun Ko and Dan Rubenstein. A distributed, self-stabilizing protocol for placement of replicated resources in emerging networks. In *Proceedings of the 11th IEEE ICNP*, 2003.

[KRS00] P. Krishnan, Danny Raz, and Yuval Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–582, 2000.

[LGI$^+$99] Bo Li, Mordecai Golin, Giuseppe F. Italiano, Xin Deng, and Kazem Sohraby. On the optimal placement of web proxies in the internet. In *Proceedings of INFOCOM*. IEEE, 1999.

[LL03] Huaiyu Liu and Simon S. Lam. Neighbor table construction and update in a dynamic peer-to-peer network. In *Proc. IEEE ICDCS*, pages 509–519, May 2003.

[LLR94] Nathan Linial, Eran London, and Yuri Rabinovich. The geometry of graphs and some of its algorithmic applications. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 577–591, 1994.

[LNBK02a] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, 2002.

[LNBK02b] David Liben-Nowell, Hari Balakrishnan, and David Karger. Observations on the dynamic evolution of peer-to-peer networks. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[LS03] Ching Law and Kai-Yeung Siu. Distributed construction of random expander networks. In *Proceedings of IEEE INFOCOM*, 2003.

[LSG$^+$04] Jinyang Li, Jeremy Stribling, Thomer M. Gil, Robert Morris, and M. Frans Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, February 2004.

[MMGC02] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, pages 31–44, 2002.

[NW03a] Moni Naor and Udi Wieder. Novel architectures for p2p applications: the continuous-discrete approach. In *Proceedings of the 15th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 50–59, 2003.

[NW03b] Moni Naor and Udi Wieder. A simple fault tolerant distributed hash table. In *Second International Workshop on Peer-to-Peer Systems*, 2003.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andr&#233;a W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the 9th Annual Symp. on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[PRU01] Gopal Pandurangan, Prabhakar Raghavan, and Eli Upfal. Building low-diameter p2p networks. In *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science*, pages 492–499, 2001.

[Räc02] H. Räcke. Minimizing congestion in general networks. In *Proceedings of the 43rd Annual Symposium on the Foundations of Comu ter Science*, pages 43–52, November 2002.

[RD01a] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware*, pages 329–350, 2001.

[RD01b] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of SOSP*, 2001.

[REG$^+$03] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *Conference on File and Storage Technologies*, pages 1–14, March 2003.

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. of SIGCOMM*, pages 161–172, 2001.

[RGRK] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. To appear in USENIX '04 Annual Technical Conference.

[RGRK03] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. Technical Report UCB//CSD-03-1299, UC Berkeley, 2003.

[RRRA99] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *Proceedings of ICDCS*, 1999.

[RRVV01] Rajmohan Rajaraman, Andrea W. Richa, Berthold Vocking, and Gayathri Vuppuluri. A data tracking scheme for general networks. In *Proceedings of the 13th Annual Symposium on Parallel Algorithms and Architectures*, pages 247–254, 2001.

[SFG$^+$02] Jared Saia, Amos Fiat, Steve Gribble, Anna R. Karlin, and Stefan Saroiu. Dynamically fault-tolerant content addressable networks. In *First International Workshop on Peer-to-Peer Systems*, 2002.

[SHK03] Jeremy Stribling, Kirsten Hildrum, and John D. Kubiatowicz. Optimizations for locality-aware structured peer-to-peer overlays. Technical Report UCB/CSD-03-1266, UC Berkeley, Computer Science Division, August 2003.

[SKKM02] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[SM02] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Proceedings of the First International Workshop on Peer-to-Peer Systems*, Cambridge, MA, 2002.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM*, pages 149–160, 2001.

[SMLN$^+$03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[TZ01] Mikkel Thorup and Uri Zwick. Approximate distance oracles. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, pages 183–192, July 2001.

[VYW$^+$02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[WWE$^+$01] Hakim Weatherspoon, Chris Wells, Patrick R. Eaton, Ben Y. Zhao, and John Kubiatowicz. Silverback: A global-scale archival system. Submitted for publication to ACM SOSP, 2001.

[Yia93] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, 1993.

[ZCB96] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM*, 1996.

[ZHR$^+$03] Ben Y. Zhao, Ling Huang, Sean C. Rhea, Jeremy Stribling, Anthony D Joseph, and John Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003. Special Issue on Service Overlay Networks, to appear.

[ZKJ01] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, 2001.