A Degree-Optimal, Ordered Peer-to-Peer Overlay Network

by

Kevin C. Zatloukal

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
[June 2005]
May 2005

Author .....
Department of Electrical Engineering and Computer Science
February 22, 2005

Certified by .....
David R. Karger
Professor of Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Degree-Optimal, Ordered Peer-to-Peer Overlay Network

by

Kevin C. Zatloukal

Submitted to the Department of Electrical Engineering and Computer Science
on February 22, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Peer-to-peer overlay networks are structures for organizing nodes, routing traffic, and searching for data in a distributed system. Two of the most important theoretical advancements in this area were the development of *degree-optimal* and *ordered* networks. Prior to this work, it was not known whether both properties could be achieved simultaneously. This thesis presents *Family Trees*, the first peer-to-peer overlay network that is both degree-optimal and ordered. We develop Family Trees theoretically, proving correctness and bounds on their performance. We also describe how Family Trees can be optimized to improve latency and discuss the results of an experimental study showing that Family Trees provide good performance in practice.

Thesis Supervisor: David R. Karger
Title: Professor of Computer Science

Acknowledgments

I would like to thank David Karger for advising and supporting me in this work.
I would also like to thank Nick Harvey for suggesting this problem, developing Family Trees along side me, carefully reviewing this thesis, and providing much feedback, encouragement, and friendship.
Finally, I would like to thank Margaret Zatloukal for her endless support, patience, and love throughout this work.

Contents

Chapter 1

Introduction

The development of peer-to-peer overlay networks was one of the most important in the last decade of research into organizing nodes in a distributed system. Accordingly, they have been given a great deal of attention. A large number of distinct systems have been developed (such as [23, 21, 22, 29, 12, 1, 13, 18]), each further advancing the state of the art in various respects. Two of the most important theoretical advancements were the development of *degree-optimal* [13, 18] and *ordered* [12, 1] networks. Prior to this work, it was not known whether both properties could be achieved simultaneously. This thesis presents *Family Trees*, which is the first peer-to-peer overlay network that is both degree-optimal and ordered.

The rest of this chapter provides background and motivation for a degree-optimal, ordered network[1]. Chapter two provides more background by describing SkipNet [12], the first ordered network with efficient lookup performance. Chapter three further motivates ordered networks by describing an application that takes advantage of their unique features. Chapter four gives the theoretical development of Family Trees, including proofs of correctness and performance bounds. Chapter five addresses Family Trees in practice, including optimizations to improve latency as well as results of an experimental study. We conclude in chapter six.

[1]Throughout the rest of this thesis, we will often shorten the long and awkward term "peer-to-peer overlay network" to simply "network".

1.1 Peer-To-Peer Overlay Networks

Peer-to-peer overlay networks are structures for organizing nodes, routing traffic, and searching for data in a distributed system. Peer-to-peer networks have numerous practical applications. The most important of these has been to the implementation of distributed hash tables (DHTs), which provide a distributed storage layer over the nodes of the network. DHTs implemented using peer-to-peer overlay networks can offer significant advantages due to the fact that all nodes are equal peers:

1. **Availability**: They are robust to individual node failures and lack single points of failure. As a result, they are also more resistant to denial-of-service attacks.

2. **Distributed Storage**: They utilize the storage space of all the individual nodes of the network. They also do so in a way that balances the storage evenly across the nodes.

3. **Low Congestion**: They spread the load of requests and associated network traffic evenly across the nodes.

DHTs implemented using peer-to-peer overlay networks clearly improve upon central storage in terms of availability and total capacity. They also improve upon other distributed storage implementations such as those using Scalable Distributed Data Structures [17], which do not guarantee low congestion. Implementations using peer-to-peer overlay networks have also provided features such as caching of documents throughout the network [6, 4], which reduces retrieval load and latency and also provides redundancy in case of node failure.

Another application of peer-to-peer overlay networks is to multicast, which is useful in many distributed systems. Multicast could be implemented directly in the Internet; however, this is would require updating numerous (hardware) routers that are already deployed. In contrast, it is easy to implement multicast in peer-to-peer overlay networks because the routing is controlled by the application (software).

Peer-to-peer overlay networks provide features such as distributed storage and multicast that would be useful building blocks for many distributed systems. Indeed,

shared storage and the ability to send messages between nodes and groups of nodes are perhaps the two fundamental building blocks of Internet applications.[2] Thus, it seems likely that, over time, peer-to-peer overlay networks will become a basic service provided by every distributed applications platform. However, before they are widely deployed, there are some additional issues that must be addressed.

1.2 Ordered Networks

Most of the prior peer-to-peer overlay networks such as Chord, Pastry, Tapestry, Koorde, and Viceroy are *unordered* in the sense that each node is identified by a key that is effectively a random number. These networks take advantage of randomness to create a routing structure over the nodes that assures that all messages are routed efficiently: in at most $O(\log n)$ hops, where n is the number of nodes in the network, with high probability.

Despite their simplicity and elegance, unordered networks are lacking in some important respects:

1. They cannot restrict the storage of documents or the network traffic for a request to stay within a given administrative domain.[3]

2. They cannot gracefully survive an administrative domain becoming disconnected from the network (i.e., a network partition). Typically, the nodes outside that domain will continue to function, albeit with significant but temporary performance degradation. However, the nodes inside that domain may be cease to function all together [12].

3. They cannot efficiently enumerate all nodes in the network within a particular administrative domain.

[2]Distributed Data Structures [10] were also motivated by the belief that providing distributed storage would make Internet applications easier to write.

[3]It is possible to achieve the storage restriction in some P2P networks, such as Chord [5], but only with extra overhead and lookup times proportional to the size of the entire network rather than the size of that domain.

All of the failures mentioned above relate to the fact that unordered P2P networks are (purposefully) ignorant of the structure of administrative domains. However, in practice, this knowledge is extremely important.

The service must be able to gracefully survive a domain being disconnected from the rest of the network, which is a common type of network failure. For example, if two internet service providers (ISPs), A and B, decide to partner in providing a DHT-based service, it would be unacceptable if A's users could not perform lookups of documents stored on A's servers because B lost network connectivity and vice versa; each ISP needs to be able to guarantee service based on its own servers. Similarly, for managing the storage and network resources, it is important for ISP A to know that certain DHT lookups will be satisfied using the resources under its control. It would also be important to be able to efficiently enumerate all clients within ISP A, for example, to send a message to all of its users or to initiate an upgrade their software.

Skipnet [12] and Skip Graphs [1] are peer-to-peer networks that can route messages based on keys from an ordered domain. In particular, they can use domain names [19] as keys. This allows these networks to fix problems 1-3 above. And since the similarity between two node's domain names (i.e, the number of matching sub-parts) is highly correlated with the network distance between the two nodes, lookups in the ordered space typically have substantially lower latency than lookups that are ignorant of network distance.

Ordered networks also have the ability to do unordered lookups, so they can provide the same distributed storage and low congestion as unordered networks. However, since they also include knowledge of administrative domains, they can provide features lacking in unordered networks, fixing problems 1-3 above. Such features will be critical for peer-to-peer overlay networks to become a basic service provided by every distributed applications platform. These additional features can also be leveraged to provide more advanced applications built on top of that service. We will see an example of such application in chapter three.

1.3 Degree-Optimal Networks

Prior to this work, unordered networks still had an advantage over ordered networks in terms of resources. The main resource needed for routing is "pointers" between nodes in the network. By pointer, we mean the collection of resources needed both to send messages and to make routing decisions. This includes memory to store the IP address of the node pointed to as well as to cache various information about the node that is useful for routing decisions (such as its ID and round-trip delay). In some real-time scenarios, it may be advantageous to maintain an open TCP socket for each pointer so that messages may be sent reliably without incurring the connection penalty for each message. An implementation may also want to send periodic pings along each pointer so that it can detect a machine failure before it causes delay to any message being routed. Thus, the resources consumed by a pointer may include network bandwidth.

Chord, Tapestry, Pastry, and SkipNet can build routing tables that contain only $O(\log n)$ pointers per node. In contrast, Koorde [13, 26][4] and Viceroy [18] have routing tables with only $O(1)$ pointers per node. These networks still guarantee that each message takes at most $O(\log n)$ hops. But since they use only $O(1)$ pointers per node they are degree-optimal. In addition to the theoretical importance of this discovery, degree-optimal networks also offer practical advantages:

1. All other things being equal, it is clearly preferable to use less resources. A reduced number of pointers means less memory usage and less maintenance traffic in the network.

2. Even when more resources are available, degree-optimal networks are preferable since they use up less of the resources for the basic protocol, thus leaving more resources available for other uses. These resources could be used to store extra pointers that reduce lookup latency and provide redundant routing paths. We

[4] A constant degree network based on de Bruijn graphs was independently and simultaneously developed in these two papers (published at the same workshop). Another such network was independently developed in [8]. For simplicity, we will use just the name "Koorde" to refer to this construction.

will see in chapter five that we can continue to reduce lookup latency as more resources are made available. A degree-optimal network can offer competitive performance to networks with higher degree when more resources are available but can still operate when less resources are available.

3. Fewer pointers means that less nodes are affected when another node joins or leaves the network (by choice or by crash). Each pointer that fails when a node crashes may cause a timeout during a later search, causing extra delay for that message.

For peer-to-peer overlay networks to become a basic service for distributed applications, the implementations will need to leave as many resources as possible available to the application. It is important to keep in mind that the application may participate in many networks at once, and of course, the application will most likely have its own needs for memory and network bandwidth.

1.4 Contributions

Prior to this work, it was not known whether it was possible to create a network that is both ordered and degree-optimal. The main contribution of this thesis is the first degree-optimal, ordered network, called Family Trees. In chapter four, we describe the structure of this network and formally prove its correctness and performance guarantees. In chapter five, we show how Family Trees can be optimized to improve latency, and we present the results of an experimental study confirming the practical benefits of Family Trees.

A secondary contribution of this thesis is the application of ordered networks given in chapter three. In particular, we will describe how ordered networks can be used to perform mobile object location better than previous systems. This application further underscores the importance of the additional features provided by ordered networks to application developers.

Chapter 2

Background: SkipNet

This chapter provides background by introducing SkipNet[1], the first ordered peer-to-peer overlay network with efficient lookup performance[2]. We will first define the routing structure of SkipNet. Then, we will see how the lookup and update operations are performed. We will mostly omit proofs as they are similar in spirit to what we will see in chapter four. Formal proofs can be found in [12].

2.1 Definitions

Each node in a SkipNet network has two identifiers. First, it has a *name ID*, a friendly name, which can be of any ordered type. Domain names like `theory.csail.mit.edu` are a common choice for name IDs. Second, each node has a *numeric ID*, which is an infinite sequence of random bits, each chosen uniformly and independently. Equivalently, we can think of the numeric ID as the binary expansion of a random real number in the range $[0, 1)$. If X is a node, then we will denote its name ID and numeric ID by X.NAMEID and X.NUMID, respectively.

As we will see in section 2.2, nodes can be looked up by either their name or

[1]The routing structure of SkipNet was independently invented in [1] and called Skip Graphs. In this thesis, we will use the name SkipNet, but we could equally well have called them Skip Graphs. However, the SkipNet work focused more on the practical aspects of the structure, which are central to our discussion of how ordered networks improve the manageability of the network.

[2]Earlier networks were able to perform ordered lookups but not with any worst-case guarantee on their performance.

numeric ID. The former operation allows nodes to send messages to each other using their friendly names. The latter operation (or rather, finding the *closest* node by numeric ID) can be used to implement a distributed hash table. For each key-value pair (K, V), we hash K into a string of pseudo-random bits, which we also interpret as a real number R in the range $[0, 1)$. We store (K, V) at the node whose numeric ID is closest to R.[3] To look up a value, we hash the key and retrieve the value from the node with closest numeric ID. The hashing function ensures that each node receives an equal fraction of all the key-value pairs in expectation. (Better guarantees can be made by using virtual nodes [14] or more sophisticated techniques such as [15]. However, these are also built on top of the lookup by numeric ID operation.)

We have defined a numeric ID to be an infinite sequence of binary digits, which we can interpret as a random real number in $[0, 1)$. We could equally well define a numeric ID to be an infinite sequence of digits in some other base. The structure and algorithms described below can all be generalized to an arbitrary base. In practice, it is common to use a higher base in order to improve performance (by a constant factor). In this thesis, however, we will stick to base 2 (binary digits) for a couple of reasons. First, base 2 is a little easier to describe and think about. Second, using a larger base increases the number of pointers per node by a constant factor. Since we are interested in structures with a small number of pointers per node (ideally, a small constant), base 2 is the most interesting case for us.

While a numeric ID can be an "infinite sequence" in principal, this would not be practical to implement. In practice, we can generate bits on demand. We will only need to have enough bits at any time so as to distinguish each node's numeric ID from that of every other node. The following proposition shows that $O(\lg n)$ bits are enough, where n is the number of nodes in the network.

Proposition 2.1.1. *With high probability*[4]*, each node in a SkipNet network needs to store only* $O(\lg n)$ *bits of its numeric ID.*

Proof. Let X and Y be nodes. The probability that X and Y choose the same first

[3] We will define "closest" in section 2.2.

[4] See the footnote in chapter four for a definition of "with high probability".

$(c+2)\lg n$ bits is $1/2^{(c+2)\lg n} = 1/n^{c+2}$. Thus, the probability that any node chooses the first $(c+2)\lg n$ bits the same as X is $(n-1)/n^{c+2} < 1/n^{c+1}$, and the probability that any node needs more than $(c+2)\lg n$ bits is less than $n/n^{c+1} = 1/n^c$. □

In order to route traffic in the network, each node must have pointers[5] to some of the other nodes. As with in-memory data structures, the layout of these pointers must be carefully designed to support efficient and correct lookup operations. Different peer-to-peer overlay networks organize their pointers in different ways, which gives each network unique properties.

The structure of SkipNet is inspired by skip lists [20]. In fact, it is important to note that any dictionary data structure, such as binary trees or skip lists, could be used to implement name ID lookup over a distributed set of nodes. However, such an implementation would have poor congestion: if we choose two nodes at random and have one lookup the other, the request would be routed through the root node[6] with probability $\frac{1}{2}$. Thus, the root node would quickly become a bottleneck and prevent further scaling. The key insight of SkipNet is to make every node be a root node so that congestion is spread evenly over the network. We will now see how this is accomplished.

In both skip lists and SkipNet, each pointer exists at a particular "level". A pointer at level i points to a node that would be roughly 2^i positions away if the nodes were in a list sorted by name ID. By following a pointer at the appropriate level, we will see that we can roughly halve the distance to the destination in one hop, which means that we can find the destination in $O(\lg n)$ hops.

In skip lists, every node has pointers at level 0. (These link the nodes into a circular doubly-linked list.) Pointers at level 1 should point two nodes away on average. A skip list creates these pointers by selecting half of the nodes at random, and linking those nodes into a list. SkipNet, in contrast, creates two lists: if bit 0 of the numeric ID is 0, the node is placed in the first list, and if it is 1, the node is placed in the second list. This way, both of the lists have pointers that point two nodes away in

[5]See section 1.3 for more information on "pointers".

[6]In a skip list, the "root node" is the node with the highest level.

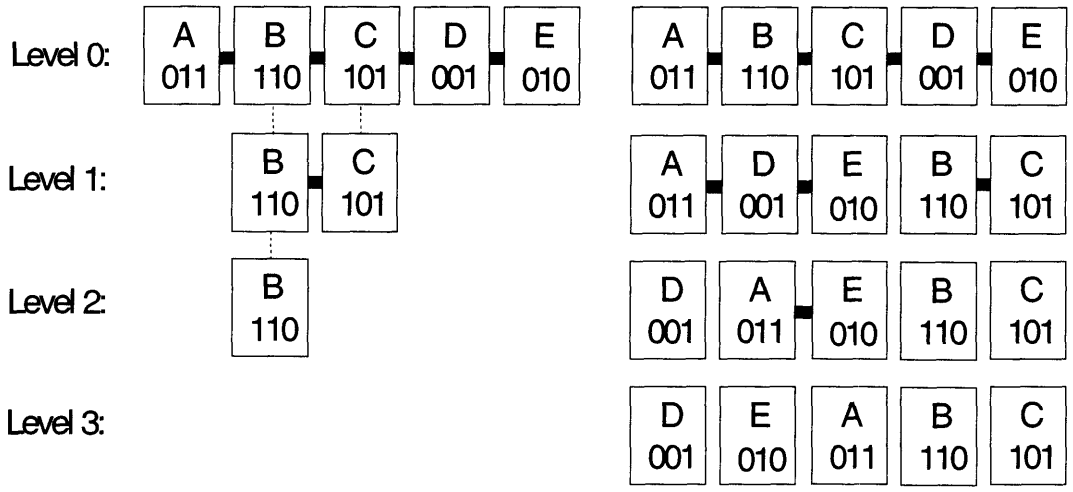

Figure 2-1: *Compares the process of building skip lists and SkipNet. Each node is a rectangle containing a name ID (letter) and numeric ID (3 bits). The dark horizontal lines show which nodes are in the same list at each level. In the skip list, at each successive level, we include only those nodes whose next numeric ID bit is* 1. *In SkipNet, we create two lists at the next level: one for those nodes whose next bit is* 0 *and one for those whose next bit is* 1.

expectation. To build a skip list, we continue taking half of the list until we get a single list containing just the root node. To build SkipNet, we continue splitting each list in half until each node is in a list by itself. Figure 2-1 demonstrates the two constructions.

In both structures, we can see that a pointer at level i points roughly 2^i positions away. However, in SkipNet, every node has pointers at high levels (every node is a root node in this sense). Thus, when a search is started at node X, we can use X's high-level pointers in order to advance half way to the destination, rather than having to use the high-level pointers of a single root node. As a result, congestion is spread evenly over the network (assuming that the source and destinations of lookups are spread evenly).

Above, we mentioned that skip lists could be used to perform lookups by name ID over a distributed set of nodes (although this would have poor congestion). Of course, we could also use SkipNet or Family Trees as an in-memory dictionary data

structure. For that reason, we have also referred to ordered networks as "distributed dictionaries" [27]. Such an implementation may be advantageous in situations with many concurrent readers and writers. In a skip list, the congestion of the root node would translate into long wait times in order to get a write lock on that node. The low congestion of SkipNet and Family Trees could potentially improve performance in this situation. However, since our focus has been on distributed implementations, we will not discuss in-memory implementations any further.

The construction of SkipNet given above described all of the necessary pointers. Let's now define them formally. Consider any node X. For each level i, X has pointers into the circular, doubly-linked list of nodes whose numeric IDs start with the same first $i-1$ bits as X's. We will denote X's pointers into this list as X.LEVELPREV[i] and X.LEVELNEXT[i]. We define the level-0 list to be sorted by name ID. And since all other lists are sublists of that one, they are sorted by name ID as well. Lastly, note that, once we get to a list where X is the only node present, there is no need to maintain any pointers. Thus, it follows from Proposition 2.1.1 that each node in a SkipNet network has $\Theta(\lg n)$ pointers with high probability. We will denote by X.LEVELMAX the last value of i for which we have pointers.

2.2 Lookup Operations

Surprisingly, the simple structure described in the previous section is capable of performing efficient lookups by either name or numeric ID. The principle behind this is the fact that each list gives partial information about both name and numeric ID. Suppose we have found our way to node X in a list at level i. We have partial information about numeric IDs because we have narrowed down the space to just those matching the first $i-1$ bits of X.NUMID. And we have partial information about name IDs because know that X is the closest node to all names between X.LEVELPREV[i].NAMEID and X.NAMEID.

The key to understanding the lookup algorithms is notice that we gain more information about numeric IDs as we move to *higher* levels and we gain more information

```
LOOKUP-BY-NAME-ID( X , value )
 1  for level = X.LEVELMAX to 0
 2  do if value ≤ X.LEVELPREV[ level ].NAMEID
 3        then return LOOKUP-BY-NAME-ID( X.LEVELPREV[ level ] , value )
 4     if X.LEVELNEXT[ level ].NAMEID ≤ value
 5        then return LOOKUP-BY-NAME-ID( X.LEVELNEXT[ level ] , value )
 6  return X
```

Figure 2-2: LOOKUP-BY-NAME-ID *finds the node whose name ID is closest to the given value. If the given value is not equal to any node's* NAMEID*, then of the nodes whose* NAMEID*s fall just before and just after the value, this will return the one whose* NAMEID *is closest to the start node.*

about name IDs as we move to *lower* levels. At the highest level, we have identified a single node that matches a given numeric ID prefix, but we have no information at all about its position by name ID. At the lowest level, we know the exact position of the node by name ID, but we have not restricted the numeric IDs at all. The principle for both lookup algorithms is the same: start at the levels where we have no information and move, one level at a time, until we have complete information. If we can advance to the next level in $O(1)$ hops, then lookup takes at most $O(\lg n)$ hops all together. Next, we will discuss each of these algorithms in more detail.

To lookup by name ID, we need to find the node whose NAMEID is closest to the value given. To do this, we will maintain the invariant that, when we are at level i, we are at the closest node in the level list to the value given. Formally, we should be at the node X such that the lookup value falls between X.LEVELPREV[i].NAMEID and X.NAMEID. It is easy to start this invariant: we pick i to be the level where the start node is in a list by itself. (Then, every value falls in the range described.) When we decrease i by 1, the invariant may be violated at the current node. To fix it, we have to move in the level list to find the closest node to the lookup value. Because the list at level i is a random sublist of the one at level $i-1$, it is not hard to show that this takes $O(1)$ hops in expectation.[7] Once we get down to level 0, we have found the destination node.

[7]Moving to the closest node may take $O(\lg n)$ hops in bad cases. We cannot guarantee less than $O(\lg n)$ hops with high probability. However, these bad cases will happen at most a small number of times over the whole lookup process, with high probability. Thus, it can be shown that a lookup takes $O(\lg n)$ hops with high probability.

Figure 2-2 shows pseudocode for this operation. It differs from what we just described in two respects. First, it does not keep track of i explicitly. Instead, each node processing the lookup request simply recomputes i, which is called *level* in the pseudocode. This is computed by finding the highest level for which the invariant does not hold. If no such level exists, then we are at the destination node. If such a level is found, then the request is forwarded toward the closest node in that level list. As mentioned above, this will take only $O(1)$ hops in expectation. Below, we will discuss the second difference between the pseudocode and our previous description. The second difference from what we just described is discussed below.

Lookup by name ID has another important property: locality. If the destination node is nearby, then *level* will be a small value initially. In particular, if the start and destination nodes are D positions apart, then the same argument from above shows that this operation takes $O(\lg D)$ hops. If D is a small value, then this can be a substantial improvement.

The locality of lookup by name ID is actually *strict locality* in the following sense. Let's denote the start and destination nodes by *dest* and *start*, respectively. If *dest*.NAMEID $\leq$ *start*.NAMEID, then every node X that receives the lookup request will satisfy

$$\textit{dest}.\text{NAMEID} \leq \text{X}.\text{NAMEID} \leq \textit{start}.\text{NAMEID}.$$

In other words, the request will never be sent to any node whose name ID is not between those of the start and destination nodes. The invariant described above assures us that *dest*.NAMEID $\leq$ X.NAMEID, and since each intermediate node node will only forward to a node that is closer the destination, we know that X.NAMEID $\leq$ *start*.NAMEID. Unfortunately, the invariant described above does not ensure strict locality when *start*.NAMEID $\leq$ *dest*.NAMEID. If we changed the invariant to require that the lookup value fall between X.NAMEID and X.LEVELNEXT[i].NAMEID, then we would get strict locality in this case but not the previous case. The pseudocode in Figure 2-2 gets strict locality in both cases by instead maintaining the invariant that, of the two nodes in the level list whose name IDs are closest to the lookup value, we

are always at the one closer to the start node.

Strict locality is extremely important from an administrator's perspective. For example, suppose that the name IDs are domain names and that the start and destination nodes are `a.mit.edu` and `z.mit.edu`, respectively. If we order domain names by comparing them lexicographically after *reversing* the sequence of parts (so `a.mit.edu` becomes `edu.mit.a`), then we know that every node whose name ID is ordered between `a.mit.edu` and `z.mit.edu` must end with `.mit.edu`. This means that every node that receives the request will also be within MIT, which gives us some assurances. First, it gives us some measure of security: none of the lookups will be seen by nodes outside of MIT. More importantly, it assures us that the message will be routed correctly as long as the MIT network is functioning and nodes within MIT are working. If Harvard loses network connectivity, users at MIT will not be affected in such lookups. Furthermore, if there is a problem with some node processing the request, it must be a node within MIT, which our administrator's can log into and fix. If Harvard nodes were failing, MIT's administrators would not necessarily be able to fix the problem.

Now, we turn to lookup by numeric ID. Here, the idea is to start at level 0, where nothing is known about numeric IDs, and move to successively higher levels until we reach the (hopefully) unique node that matches the most bits of the lookup value.[8] To increase the level by 1, we need to find a node X in the level list such that X.NUMID$[i] = value[i]$, where "$[i]$" denotes the ith bit. With probability $\frac{1}{2}$, the node we are already at will satisfy this. Otherwise, we need to forward to another node. The expected number of hops needed to find a node whose numeric ID that matches the next bit of the lookup value is $O(1)$.[9]

Figure 2-3 shows pseudocode for this operation. As before, we do not maintain the level i explicitly. Instead, each node computes this by calling MATCH-LENGTH,

[8]Most peer-to-peer overlay networks route to the node whose numeric ID is closest modulo 1 to the given value. It is not necessary to route to this node. All that is required is that we always route to the same node for a given value and that we spread the load evenly. The technique described here has both properties.

[9]Again, we cannot guarantee less than $O(\lg n)$ hops with high probability. However, the $O(\lg n)$ cases will occur rarely. Thus, it can be shown that, over the course of whole lookup, at most $O(\lg n)$ hops occur all together.

```
LOOKUP-BY-NUM-ID( X , value )
 7  level ← MATCH-LENGTH( X.NUMID , value )
 8  return LOOKUP-IN-LEVEL-LIST( X , level , value , nil )

LOOKUP-IN-LEVEL-LIST( X , level , value , list )
 9  if X.NUMID [ level ] = value [ level ]
10     then return LOOKUP-BY-NUM-ID( X , value )
11  if X = FRONT( list )
12     then return CLOSEST( list , value )
13     else  list ← ADD-TO-BACK( list , X )
14           return LOOKUP-IN-LEVEL-LIST( X.LEVELNEXT[ level ] , level , value , list )

MATCH-LENGTH( value1 , value2 )
15  for i = 0 to ∞
16  do if value1 [i] ≠ value2 [i]
17        then return i
```

Figure 2-3: LOOKUP-BY-NUM-ID *finds the node whose numeric ID is closest to the given value. The closest node is chosen from those nodes whose numeric IDs match the most bits of the value. The deterministic procedure to choose from amongst these is encapsulated in the function* CLOSEST.

which counts the number of bits that match, and stores the result in the variable *level*. Next, it calls LOOKUP-IN-LEVEL-LIST, which tries to find a node matching the *level*-th bit of *value*. If one is found, then the request is forwarded to that node. In case one is not found, LOOKUP-IN-LEVEL-LIST builds up a list (in the variable called *list*) of all the nodes in this level list. Once we get back to the first node in the list, we know that we have a complete list of all the nodes that match the maximum number of bits of *value*. The call to CLOSEST deterministically chooses one of these to return.[10]

Unlike name ID lookups, those by numeric ID have no locality whatsoever. Since the numeric IDs are chosen uniformly at randomly, the destination is chosen uniformly at random. For a given start node, the expected distance to the destination is the average distance from that node to all others. If the network has nodes spread out all over the world, it could take a few seconds on average to send a message to the destination node.[11] (This is even ignoring the time required for all of the extra $O(\lg n)$

[10] Again, we only require that it choose deterministically and spread the load evenly. The standard technique of picking the closest modulo 1 will work fine here. That is, we pick the node with the largest numeric ID that is less than or equal to *value*, unless there are no such nodes, in which case we pick the node with the largest numeric ID overall.

[11] Though this will undoubtedly improve as network connectivity across the world is increased, the

hops performed during the lookup.) It would be better if there were a way to restrict the numeric ID lookup to a subset of nodes that are not too far away. This is indeed possible, as we will see below.

Perhaps the most powerful lookup operation on an ordered network is one that combines both name and numeric IDs. Specifically, we can perform a lookup by numeric ID but *constrained* to a specific the subset of nodes whose name IDs fall within a certain range.[12] For example, we could constrain the search to only those nodes with names that end with `mit.edu`. Even though this may be a large number of nodes, the total lookup time will be substantially faster than if we had to send messages around the world.

The key to performing a name-constrained numeric ID lookup is to notice that, if we removed all nodes outside of the given range, we still get a perfectly valid SkipNet. In fact, we get the same distribution of possible shapes for the network of nodes within the range regardless of whether the other nodes are present. (The only minor difference is that the code above assumed circularly linked lists. We need to use non-circular lists for property to hold. But that adds only minor complication to the algorithms from above.) Thus, to perform a name-restricted numeric ID lookup, starting from a node within the range, we just perform the normal numeric ID lookup but pretend that any nodes outside the range aren't there! If we start at a node outside the range, then we first perform a name ID lookup to find a new starting node that is within the range.

We have seen that SkipNet can perform lookups by both name and numeric ID. Both operations are simple and have guarantees of $O(\lg n)$ hops with high probability. The name ID lookups have strict locality to the name ID range between start and destination nodes. Numeric ID lookups have no locality normally, but with SkipNet, they can be restricted to just the nodes of a particular domain. These locality properties impact both the performance and the administrative manageability of the network. It is also possible to show that these operations spread the load from lookup

fact that the speed of light is bounded means that sending messages across the globe will always be much more expensive than sending messages to nearby nodes.

[12] In SkipNet, this is called "constrained load balancing".

requests evenly across the network. (We will see proofs of this for Family Trees in chapter four.) In the next section, we will see how nodes can join and leave the network.

2.3 Update Operations

To join a new node Z into the network, we need to fill in the values of Z's pointers and update any nodes that now need to point to Z. Conceptually, all we are doing is linking Z into each of the level lists in which it belongs. To find Z's place in the level 0 list, we simply perform a LOOKUP-BY-NAME-ID. Next, we find Z's place in the level 1 list for nodes whose numeric IDs start with Z.NUMID[0]. Then, we continue moving up to the level list that matches the next bit of Z.NUMID until we get to a list containing Z alone.

We have seen already this process of moving upward, one level at a time, into the list that matches a particular bit: this is exactly what we did in LOOKUP-BY-NUM-ID. It is important to note that, each time we move up a level, we will reach one of the two closest nodes to Z.NAMEID in that list. This holds because we started at the closest node in the previous level and moved up at the first node encountered that matched in the next bit. Thus, the JOIN procedure is identical to numeric ID lookup on the bits of Z.NUMID except that, each time we reach a new level, we link Z into the linked list. The same arguments as for numeric ID lookup show that this procedure requires $O(\lg n)$ hops with high probability.

Pseudocode for this operation is shown in Figure 2-4. JOIN-LEVEL links Z into a level given one of the nodes closest to Z. One tricky part is lines 20–23, which set X to be the node before Z and Y the node after. Lines 24–26 simply link Z into this list. Then, we call JOIN-NEXT-LEVEL to find one of the closest nodes to Z in the next level. This proceeds just as in LOOKUP-BY-NUMERIC-ID. Once we find that we have circled the level list, we know that the next higher level list we are searching for is empty, at which point the JOIN is completed.

To delete a node X from the network, we just need to remove it from all of

```
JOIN( X , Z )
18  X ← LOOKUP-BY-NAME-ID( X , Z.NAMEID )
19  return JOIN-LEVEL( X , Z , 0)

JOIN-LEVEL( X , Z , level )
20  if Z.NAMEID < X.NAMEID or
21     X.LEVELPREV [ level ].NAMEID < Z.NAMEID
22     then X ← X.LEVELPREV [ level ]
23  Y ← X.LEVELNEXT [ level ]
24  Z.LEVELPREV [ level ] ← X
25  Z.LEVELNEXT [ level ] ← Y
26  X.LEVELNEXT [ level ] ← Y.LEVELPREV [ level ] ← Z
27  return JOIN-NEXT-LEVEL( X , Z , level , nil )

JOIN-NEXT-LEVEL( X , Z , level , list )
28  if X.NUMID [ level ] = Z [ level ]
29     then return JOIN-LEVEL( X , Z , level + 1)
30  if X = FRONT( list )
31     then return Z
32     else  list ← ADD-TO-BACK( list , X )
33           return JOIN-LEVEL( X.LEVELNEXT[ level ] , Z , level , list )
```

Figure 2-4: JOIN *adds the new node* Z *into the network. The request starts at the node* X *already in the network.*

```
LEAVE( X )
34  for level = 0 to X.LEVELMAX
35  do X.LEVELPREV [ level ].LEVELNEXT ← X.LEVELNEXT [ level ]
36     X.LEVELNEXT [ level ].LEVELPREV ← X.LEVELPREV [ level ]
```

Figure 2-5: LEAVE *removes node* X *from the network by unlinking it from each level list of which it is a member.*

the level lists to which it belongs. This simply requires, for each level i, updating X.LEVELPREV[i] and X.LEVELNEXT[i] to point to each other instead of to X. For the sake of completeness, the pseudocode of this operation is shown in Figure 2-5. It is easy to see that this procedure requires $O(\lg n)$ messages with high probability.

2.4 Conclusions

In this chapter, we have seen the design of SkipNet, the first ordered peer-to-peer overlay network with efficient lookup performance. We have seen how to perform

lookups by name ID and numeric ID and how to perform a name-constrained lookup by numeric ID. We have also seen how nodes can join and leave the network. All of these operations are simple, elegant, and efficient, requiring $O(\lg n)$ hops with high probability.

We can now see how SkipNet resolves each of the problems with unordered networks that were identified in chapter one:

1. SkipNet can constrain the storage of documents to a particular domain by choosing the host at which to store the document using name-constrained numeric ID lookups. And as mentioned above, it also guarantees that the lookup request will only be routed through nodes within that domain, which means that the lookup request will only generate traffic in the underlying network of that domain (i.e., only its routers and cables will be used).

2. SkipNet can gracefully survive a network partition. Lookups for names inside the disconnected domain will not notice the failure at all since the lookups are only routed through notes whose names are between those of the start and destination, all of which are inside the disconnected domain. Lookups for numeric IDs that are constrained to that domain will also not notice the failure for the same reason.

3. SkipNet can efficiently enumerate the nodes within a particular domain. This can be done by looking up a name in that domain and then iterating through the level-0 list. However, it is also possible to multicast to all of the nodes in a domain by applying the standard skip list range query algorithm to SkipNet. Thus, we can send a message to all nodes in a particular domain in the time $O(\lg R + \lg D)$ where R is the distance (in the level-0 list) of the start node from the nearest node in that domain and D is the distance from one side of the domain to the other.

As discussed in chapter one, these features significantly improve the manageability of the network for administrators. Thus, ordered networks with these features are much more likely to see substantial deployment in practice.

One advantage that unordered networks have over SkipNet is the amount of resources required. While each SkipNet node requires $\Theta(\lg n)$ pointers, the unordered Koorde and Viceroy networks require only $O(1)$ pointers per node. In chapter four, we will introduce Family Trees, which are an ordered network that uses only $O(1)$ pointers per node. But first, in chapter three, we further motivate ordered networks by looking at an application that takes advantage of the name-constrained numeric ID lookup, which is not available on unordered networks.

Chapter 3

Application: Mobile Entity Location

3.1 Introduction

A mobile entity location system is a distributed system that maintains a name to location mapping. The names can come from an arbitrary domain (not necessarily ordered). The locations are assumed to be positions in a hierarchical network of some sort. Clients can notify the location system of their own location, which may be changing frequently, and they can query the location of other clients. We assume that clients can use the location identifier that is returned by a query to send a message to the location in the network to which it refers.

It is fairly easy to come up with examples where such systems would be useful. In a cellular phone network, we might want a location system where the names are phone numbers and the locations are the network addresses of cell towers. In order to call another cell phone, we need to query the system for the location of phone whose number we want to call (i.e., which cell tower it is at). Once we have the location, we can send messages to that phone in order to initiate a phone call.

We might also want such a system for a peer-to-peer phone and instant messaging application like Skype [2]. In this system, clients that are behind firewalls cannot receive incoming connections from other clients. To overcome this limitation, the

client connects to a nearby "super-node", which is a client that is not behind a firewall. The firewall will allow two-way communication between the client and super-node because the client initiated that connection. To call a client that is behind a firewall, we look up the domain name of the super-node to which it is connected in a location system. We send the message to the super-node (who can receive connections from arbitrary addresses), who will then forward it to the client behind the firewall.

Location systems differ from basic distributed hashtables because they have very strong locality requirements. Specifically, if the client performing the query is in the same subdomain as the location that the query would return, the system must not send messages outside of that subdomain. In a distributed hashtable, the record for a given client could be stored on the other side of the world from their actual location. If another client tried to look up that clients location, they might have to wait several seconds for the query to complete, which is unacceptable if the client is nearby the location being queried.

For example, in a cell phone network, users expect a local call to be connected instantaneously, whereas they understand that an international call to the other side of the earth may have a noticeable delay. The cellular network is likely to have a hierarchical organization to it, which is directly reflected in location identifiers. In the past, phone numbers were location identifiers and reflected the hierarchy of the underlying phone system: each number consisted of a country code, area code, three-digit part, and four-digit part, each of which corresponded to a physical space. To allow for mobile entities like cellular phones, we separate the names (phone numbers) from location identifiers and introduce a location system to map between them. However, the user still has the expectation that a call to another user in the same physical region will be completed very quickly. Furthermore, strict locality to physical parts of the phone network is helpful to administrators of the network who must allocate physical resources in proportion to the number of customers in a given area.

For Skype, we can use the domain names of the super-nodes as locations. While these are not guaranteed to match the underlying location, they often correlate well. Domain names often identify the country correctly, and sometimes even more than

this. For example, two clients with Verizon DSL are likely to be given domain names that reflect the local region in which they reside. (And the fact that they are both Verizon customers identifies the general part of the country they are in.) Thus, locality using domain names is likely to have good response times for users. However, it would also be possible to build our own system for determining hierarchical network locations. For example, we could determine the distances between a large set of core network routers and approximate these distances with a tree metric [7]. Then, we could assign each client a location by using the location in the tree of a nearby router (which we could find by performing a trace-route to a server across the network). Locality with these locations is likely to correlate strongly with low latency. Many other techniques are possible for generating hierarchical locations. In the remainder of this chapter, we will assume that each client knows its location.

3.2 Design

A simple location system that meets the strict locality requirement works as follows [24]. We have a set of servers that are organized as a tree. The leaf nodes in the tree are servers where clients can be located. Each hierarchical location identifier of the form $a.b.c.\cdots$, identifies a leaf node by the path from the root to that node: from the root, we descend to child a, then to a's child b, then to b's child c, and so on. Figure 3-1 shows part of a tree with two clients: N at $a.b.c$ and M at $a.e.f$. At each server, we maintain a map that records, for each client located in the subtree rooted at that server, the name of the child node whose subtree it is in. For example, since client N is located at $a.c$, the root's map has N mapped to a, a's map has N mapped to b, and b's map has N mapped to c. Server c also knows that N is located there and records the relevant information about how to communicate with N.

If client M tries to find N, the query starts at $a.e.f$. This node checks its map to see if N is located there as well, but since it is not, it forwards the request to its parent, $a.e$. Server $a.e$ looks in its map, and since N is not found, forwards to its parent, a. Server a looks in its map and finds N mapped to b, so it forwards the

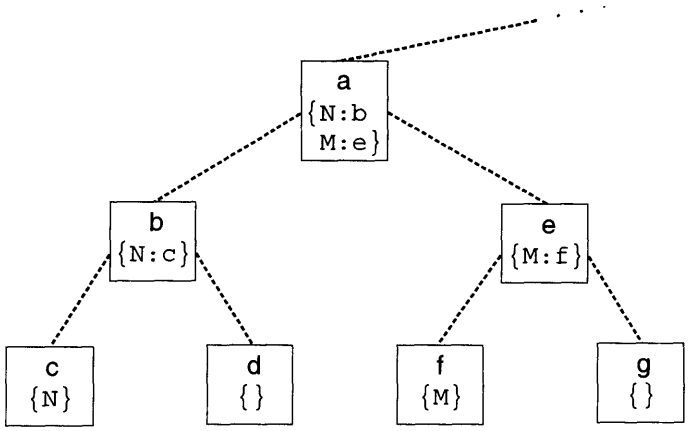

Figure 3-1: *Shows the subdomain of location tree containing locations starting with a. This subdomain contains two clients: client N is at location a.b.c and client M is at location a.e.f. The root node (the parent of a) is not shown.*

request to its child $a.b$. Server $a.b$ looks in its map and finds N mapped to c, so it forwards the request to its child $a.b.c$. Server $a.b.c$ finds that N is located there, so it sends its own location $a.b.c$ back to the source node $a.e.f$, who can use this to send a message to $a.b.c$. In this case, clients N and M are both in subdomain a, and we can see that no messages were sent outside of that subdomain. It is easy to see that this holds true generally, so we have the strict locality that is required.

When a client moves locations, we perform a similar process. The server for the new location sends the request upward until it reaches a server whose map contains that name. The server forwards the request down toward the old location, so that each server along that path can remove the name from its map. Then the server updates its map to contain the name of the child from which the update came. Each server that forwarded the request to its parent does the same. We can see that updating a client's location takes the same amount of time as performing a lookup of the client's old location from its new location. Since clients typically move only a short distance away, updates are usually very efficient.

The main problem with this simple design is that it is not scalable. Since the root node has a map containing every client's name, it will quickly run out of space. Furthermore, if there are many "long distance" queries, the root node will quickly become a bottleneck.

Previous systems have fixed this problem by replacing the servers at higher levels with clusters of servers. The problem with this approach, however, is that it requires careful layout of physical resources. The administrators of the network need to carefully watch the use of the network and add more machines to various clusters before they become bottlenecks. Aside from the management headache this causes, it is also unable to keep up if client behavior shifts quickly.

We can solve this problem by using distributed hash tables. Specifically, each location server participates in a DHT for each prefix of its hierarchical location identifier. For example, the server with $a.b.c$ participates in three DHTs: one for prefix $a.b$, one for prefix a, and one for the empty prefix. Each DHT maps client names to the next part of the location identifier of that client. For example, if client N is at $a.b.c$, then the DHT for prefix $a.b$ maps N to c, the one for a maps N to b, and the one for the empty prefix maps N to a.

This solves the scalability problem because the number of nodes in each DHT is equal to the number of locations. Since each location can support some maximum number of users, the number of names stored in each DHT should be proportional to the number of nodes (i.e., the number of names is $\leq Cn$, where n is the number of nodes in the DHT and C is the maximum number of clients per location). Furthermore, the system scales naturally on its own. As we add more locations, the affected DHTs each get more nodes.

This approach also maintains strict locality. The DHT for a given prefix contains only nodes whose locations begin with that prefix, so all messages sent to perform the DHT lookup will stay within that prefix.

The problem with this approach is that it is less efficient. For example, suppose that the location hierarchy is a perfect tree of height k. Now suppose that we perform a lookup from the very first location for a client at the very last location. Then, we will perform lookups over subdomains containing 1, 2, 4, ..., 2^{k-1}, 2^k, 2^{k-1}, ..., 2, and 1 nodes. The total number of hops needed to perform this lookup is $O(\sum_{i=1}^{k} \lg 2^i) = O(k^2)$. Whereas, in the original system, it was only $O(k)$ hops.

We can improve the performance of this system by using SkipNet, an ordered

network. In a SkipNet implementation, the name IDs are the hierarchical locations. To perform a DHT lookup of name N, we hash N to a bit string B and perform a numeric ID lookup on B. This request will be routed to the node whose numeric ID is closest to B, and this node will record the value corresponding to key N (if N is in the map). Instead of maintaining multiple separate DHTs, we can put all servers into a single SkipNet network and implement each individual lookup as a name-constrained numeric ID lookup. For example, to look up N in the subdomain $a.b$, we perform a numeric ID lookup on the hash of N that is constrained to be within nodes whose name IDs begin with $a.b$. SkipNet will find the node within that subdomain whose numeric ID is closest to the hash of the key, which is where the value would be stored. Furthermore, it will honor the name constraint, and thus maintain strict locality.

The key to improving the performance of the lookups is the following observation. Suppose that we are performing a lookup for N starting from location $a.b.c$. We first perform a lookup constrained to $a.b$ and then a lookup constrained to a. In the first lookup (constrained to $a.b$), we move upward through the levels, ignoring nodes not in subdomain $a.b$ until we reach a level list containing only a single node. This is where the location of N would be stored if that client were located in this subdomain. Suppose that N is not found. Our next lookup would be for the same key but would expand the name constraint to all of a. If we started a new lookup back from our original location, it would most likely follow exactly the same path as the last search. The key point is that we can just as well start off the next search from where the last one ended since that node satisfies the name constraint. Once we have performed the lookup for the largest subdomain, where N is finally found in the map (in this case, a), if we look back over the sequence of lookups, it would look exactly the same as if we had just performed a single lookup constrained to that largest subdomain. So rather than performing $O(\lg^2 n)$ hops (where n is the number of nodes in the largest sub-domain), we have only performed $O(\lg n)$.

Actually, there is one minor difference between this sequence of lookups and a single lookup constrained to the largest sub-domain. For each subdomain, at the highest level, there may have been multiple nodes, in which case, we would have

had to walk through this list to find the closest matching numeric ID. However, this is only $O(1)$ hops per level in expectation, so the total number of hops for the whole sequence is still $O(\lg n)$ in expectation. Furthermore, even though we can only guarantee $O(\lg n)$ hops at the highest level of each subdomain with high probability, we can guarantee that, over the whole sequence of lookups the total number of hops is $O(\lg n)$ with high probability.

Once we have found a mapping for N, we now need to perform lookups that are more constrained. Imagine that, instead of starting at $a.b.c$ looking for the location of N, say $a.e.f$, we had started at $a.e.f$ and looked for the location of N, which was $a.b.c$. In that case, we would move upward through the levels, expanding the name constraint each time N is not found, until we reach the destination for the lookup constrained to subdomain a. The key is to notice that we can perform the downward lookups (increasingly restrictive) just by simulating the upward sequence in reverse. Each time we add a part to the name constraint, we need to move down levels until a node satisfying that constraint is found. At each level, we find the node that is closest to satisfying the constraint, and if it does not satisfy the constraint, we move down another level. Thus, we can see that this will take $O(\lg n)$ time as well, with high probability.

We have now seen that we can use SkipNet to implement a mobile entity location system. It can scale gracefully and automatically as new locations are created. This location service is easy to implement using name-constrained numeric ID lookups. As we have seen, they are a very useful operation in practice. Furthermore, we showed how a sequence of expanding or contracting set of name constraints with hierarchical names can be optimized to perform as just a single lookup. The resulting system achieves $O(\lg n)$ hops per query, where n is the number of nodes in the smallest subdomain containing both the source and the destination locations.

Chapter 4

Family Trees

In this chapter, we introduce Family Trees[1], the first peer-to-peer overlay network that is both degree-optimal and ordered. We begin by defining the routing structure of Family Trees. Then, we will examine some of their global properties. Finally, we will show the algorithms for lookups and updates to the network and provide formal proofs of their correctness and performance.

4.1 Definitions

As we saw in chapter two, each SkipNet node is linked into lists at $\Theta(\lg n)$ different levels. To reduce the number of pointers to $O(1)$, Family Trees link each node into only one of these list. The resulting structure also resembles Viceroy [18] and the butterfly network [16] on which it is based. As in Viceroy, nodes in Family Trees are separated into approximately $\lg n$ levels; nodes at level i will have pointers to nodes approximately $2^i \lg n$ positions away in the ordered list of names. However, we generate these pointers by separating the nodes at level i randomly into 2^i separate ordered lists, in a manner similar to SkipNet. The resulting data structure has a natural analogy to a genealogical family tree, as will be made clear shortly. Figure 4-1 shows an example instance of a family tree.

As in SkipNet, each Family Tree node has name and numeric IDs. Next, each

[1] An earlier version of this chapter was published in [27].

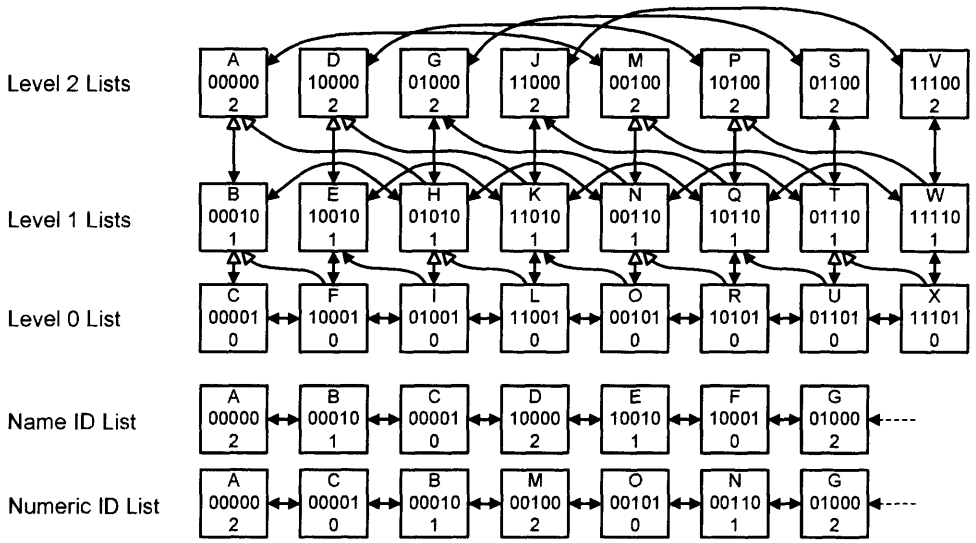

Figure 4-1: *An example family tree with 24 nodes. Each node, denoted by a square, has a name ID, numeric ID, and level. Level 0 nodes are connected into a single list. The level 1 nodes are divided into two disjoint, interleaved lists: nodes whose numeric IDs start with a 0, and nodes whose numeric IDs start with a 1. At level 2, there are four lists: nodes join the list given by the first two bits of their numeric ID. All of these level lists are sorted by the name IDs of the nodes. Nodes also maintain pointers to their parents, one level higher: white pointers denote a mother, and black pointers denote a father. Nodes also point to their first child, one level lower. Lastly, all nodes belong to a list sorted by name IDs and a list sorted by numeric IDs.*

node X has a *level*, denoted X.LEVEL. As with SkipNet, there are 2^i different lists at level i. X will be linked into just one of these lists, the one containing nodes that match the first X.LEVEL bits of X's numeric ID. Each node X chooses X.LEVEL uniformly at random from $\{0, \ldots, \lfloor \lg n_0 \rfloor - 1\}$, where n_0 is an estimate of n. We must use an estimate for two reasons. First, it is unlikely that n can be precisely computed in a manner that is efficient in both time and congestion. Second, it is helpful that different nodes produce different estimates of n so that nodes are not all alerted to the presence of a new level simultaneously: if we computed n exactly, then every node would need to update its level when n reached the next power of 2.

We will discuss the method of estimating n in section 4.2. In the remainder of this subsection, we will define all of the pointers on a node. These pointers are the only resources required by each node beyond its name ID, numeric ID, and level.

Each Family Tree node X has nine pointers. The first six pointers link each node

into three circularly linked lists. The X.NAMEPREV and X.NAMENEXT pointers link X into a list containing *all nodes* sorted by name ID. The X.NUMPREV and X.NUMNEXT pointers link X into a list containing all nodes sorted by numeric ID. The X.LEVELPREV and X.LEVELNEXT pointers link X into the single level list in which it participates: that is, the list at level X.LEVEL containing nodes that have the same first X.LEVEL bits in their numeric IDs as X. As in SkipNet, each level list is sorted by name ID.

Additionally, there are pointers between levels. X.MOTHER points into the list of nodes with level equal to X.LEVEL + 1, with numeric ID matching the first X.LEVEL bits of X.NUMID, and with (X.LEVEL + 1)-th numeric ID bit equal to 0. Of these nodes, X.MOTHER points to the node whose name ID is closest but *less than* X's, or it points to NIL if no such node exists. X.FATHER is defined identically except its (X.LEVEL + 1)-th bit must be 1.

Lastly, X.FIRSTCHILD points into the list of nodes in level X.LEVEL − 1 whose numeric IDs match the first X.LEVEL − 1 bits of X.NUMID. Amongst these nodes, X.FIRSTCHILD points to the node whose name ID is closest but *greater than* X's, or it points to NIL if no such node exists.

For example, consider the node with name ID "N" in Figure 2-1, which we will denote N. Node N is linked into the list sorted by name ID and the list sorted by numeric ID. Since N has chosen level 1, it belongs to one of the two level 1 lists, according to the first bit of its numeric ID. This bit is 0, so N belongs to the same level 1 list as "B", "H", and "T". The first child of N is "O" because this is the next name after "N" in the sole level 0 list. Node "R" is also a child of N. Node "M" is the closest node to "N" in the level 2 list corresponding to numeric ID prefix 00, so node "M" is N.MOTHER. Similarly, node "G" is N.FATHER as it is the closest node to "N" in the level 2 list with prefix 01.

4.2 Estimating n

In order to choose a level, each node must estimate n. For this, we use the estimation procedure of Viceroy. Let X be a node and Y be its successor in numeric ID space. (The last node in the list will take the first as its successor.) Then, regarding the numeric IDs as real numbers in $[0, 1)$, we estimate n as $n_0 = 1/d(\text{X.NumID}, \text{Y.NumID})$, where $d(x, y) = y - x \mod 1$. The expected value of $d(\text{X.NumID}, \text{Y.NumID})$ is $\frac{1}{n}$ since there are n nodes in the network, so the average value of n_0 should normally be close to n. However, this estimate may be substantially incorrect. Thankfully, we only need the log of this estimate. We will now show that every node estimates $\lg n_0 = \Theta(\lg n)$ with high probability.[2]

Proposition 4.2.1. *Every node estimates n_0 such that $\lfloor \lg(n/(c+2)\ln n) \rfloor \leq \lfloor \lg n_0 \rfloor \leq \lfloor (c+2)\lg n \rfloor$, for any constant c, with high probability.*

Proof. First, we will argue the lower bound. Let $d = 1/(n/(c+2)\ln n) = (c+2)\ln n/n$. For a node X to estimate n smaller than $1/d$, every other node must have chosen a numeric ID outside of the range of length d after X.NumID.[3] The probability that X estimates n this small is $(1-d)^{n-1} = (1-d)^n/(1-d) < n(1-d)^n < n\exp(-dn) = n\exp(-n(c+2)\ln n/n) < n\exp(-(c+2)\ln n) = 1/n^{c+1}$. Thus, the probability that any node estimates n this small is less than $n/n^{c+1} = 1/n^c$.

Now, we argue the upper bound. Let $d = 1/n^{c+2}$. Fix a node X. The probability that some other node Y has a numeric ID within the range of length d after X.NumID is d. Thus, the probability that X estimated $n > 1/d$ is at most $(n-1)d = (n-1)/n^{c+2} < 1/n^{c+1}$. Thus, the probability that any node estimated $n > 1/d$ is less than $n/n^{c+1} = 1/n^c$.

We have shown that the desired bounds hold without the floors. Taking the floor of the estimate and both bounds can only increase the probability that the bounds hold (if it was true before, then it is true with the floors), so the proof is complete. □

[2]Throughout this thesis, we say that X is "$O(f(n))$ with high probability" to mean that there exists an $\alpha > 0$ such that $\Pr(X > c\alpha f(n)) < 1/n^c$ for any $c \geq 1$ and for sufficiently large n.

[3]If X.NumID is near to 1, then this range will be in two pieces: one at the end of $[0, 1)$ and one at the beginning.

An advantage of this method for estimating n is that each node's estimate is only dependent on the distance to its successor in the numeric ID list. Thus, each node needs to change its estimate only when its successor changes, which will make insertions and deletions efficient. We will discuss this aspect further in section 4.5.

A notable disadvantage of this method is that it makes the levels dependent on the numeric IDs. Furthermore, the levels themselves are not independent. If X has a level of i, then we know that there is some node whose numeric ID falls at most $1/2^{i+1}$ after its own. This means that some other node estimated $n_0 \leq (n-1)/(1-1/2^{i+1})$ and thus has a level at most $\lfloor \lg (n-1)/(1-1/2^{i+1}) \rfloor - 1$. Clearly, this is a small amount of dependence: we have only shown that some node estimated n_0 to be slightly less than what we would expect to be its average value. However, even this small amount of dependence complicates analysis. And as we have seen the levels of more and more nodes, the amount of knowledge about the levels of the remaining nodes increases. This issue of dependence also exists for Viceroy, although it was not discussed explicitly in their analysis.

To handle this difficulty, we use the following approach. Proposition 4.2.1 shows that every node X estimates n reasonably with high probability, where reasonably means that $\Pr(\text{X.LEVEL} = i \mid \text{X estimated } n \text{ reasonably})$ is within a constant factor of $1/\lg n$. This yields a bound on $\Pr(\text{X.LEVEL} = i)$ as follows. In general, if E is some event that holds with high probability and F is any other event, we can bound $\Pr(F)$ using $\Pr(F|E)$ and an additional error term. Specifically, we have $|\Pr(F) - \Pr(F|E)| \leq 1/n^c$. As long our probability and expectation bounds introduce no more than n^α such error terms, for some fixed α, we can use $\Pr(F)$ and $\Pr(F|E)$ interchangeably while only introducing an error term of $1/n^{c-\alpha-1}$. By choosing suitably large c, this error term disappears in the O-notation. In the analysis below, we can assume that all nodes estimate n within the bounds of Proposition 4.2.1 whenever convenient because we will not need to introduce more than $O(n)$ error terms. Thus, we will ignore error terms in the proofs below in order to keep the explanations clear.

To simplify notation, we will let the c in Proposition 4.2.1 be fixed and define the following notation for the upper and lower bounds.

Definition 4.2.1. The minimum level estimate is $L_{\min} = \lfloor \lg(n/(c+2)\ln n) \rfloor$ and the maximum level estimate is $L_{\max} = \lfloor (c+2)\lg n \rfloor$.

4.3 Global Properties

Now that all data held by each node has been defined, we examine some global properties of Family Trees.

Proposition 4.3.1. *If all name IDs, numeric IDs, and levels are given, then the Family Tree has only one possible shape.*

Proof. Since we do not allow duplicates, the name ID list has only one possible shape. Similarly, the numeric ID list has only one possible shape. Each node belongs to exactly one level list according to its level and the relevant bits of its numeric ID. Each such list is sorted, so they can have only one shape. Lastly, the MOTHER, FATHER, and FIRSTCHILD pointers are completely determined by the shape of the level lists. □

Corollary 4.3.1. *The probability that a Family Tree has a given shape is independent of history.*

Proof. By Proposition 4.3.1, the current shape of the data structure depends only on the name IDs, numeric IDs, and levels of the items currently in the dictionary. The name IDs and numeric IDs themselves are clearly independent of history. As long as each node chose its level by estimating n using its current successor (not a past successor), then the probability distribution of the levels is independent of history. We will ensure that this is true by having a node choose a new level whenever its successor changes. □

The above properties are important because they imply that we can analyze each operation on the Family Tree independently of all the operations that occurred before. If this were not the case, we would have to consider all sequences of operations in our analysis, which would complicate matters significantly.

Next, we turn to properties of Family Trees that will be useful in analyzing the performance of search and update operations. For all operations, it is necessary to bound the number of levels in the data structure since it may be necessary to visit all of them. Thanks to our earlier proposition, we can bound this very tightly.

Theorem 4.3.1. *The data structure has no more than* $(c+2)\lg n$ *levels with high probability.*

Proof. By Proposition 4.2.1, no node estimated $\lg n_0$ larger than this. Hence, no node could have chosen a level larger than this. □

Since the nodes at level L are separated randomly into 2^L lists, the expected length of these lists is roughly $n/2^L \lg n$.[4] The following result shows that, at all but the highest levels, the actual length is within a constant factor of this expected length with high probability.

Theorem 4.3.2. *Every list at level* $L \leq \lg n - 2\lg\lg n - \lg(c(c+2))$ *is of length* $\Theta(n/2^L \lg n)$ *with high probability. Every list at a higher level is of length* $O(\lg n)$ *with high probability.*

Proof. We first look at levels $L \leq \lg n - 2\lg\lg n - \lg(c(c+2))$. We will prove the lower bound; the upper bound is similar. Let μ be the expected length of the list. For any L in this range, we can see that $\mu \geq n/2^L L_{\max} = c\lg n$. Then, for any $\delta \in [0,1)$, we have $\Pr(X < (1-\delta)\mu) < 2^{-\delta^2\mu/2} < 1/n^{\delta^2 c/2}$, by the Chernoff bound.

For levels higher than this, the Chernoff bound is not useful because the mean is $o(\lg n)$. However, it is clear that the probability of being in such a list is smaller than the probability of being in the list at level $\lg n - 2\lg\lg n$. Thus, we can bound the length of those lists by the length of the list at $\lg n - 2\lg\lg n$, which is $O(\lg n)$ as we saw above. □

An important issue for our algorithms is whether the list at a given level is empty or not. Empty lists could be problematic because they mean that any higher levels (if present) are disconnected from other levels (since their child pointers are NIL).

[4]Recall again that we are ignoring lower order terms due to the dependency between levels.

The following theorem shows that we can rule out the possibility of empty lists for all levels up to nearly $L_{\min}$.

Theorem 4.3.3. *Let $L = \lg n - 2\lg\lg n - 2\lg(c+2)$. Every list in the family tree with level at most L is non-empty with high probability.*

Proof. The probability that a list at level k is empty is at most $(1-1/(2^k(c+2)\lg n))^n$. If $k \leq L$, then this bound is at most $(1-(c+2)\lg n/n)^n < e^{-(c+2)\lg n} = 1/n^{c+2}$. Next, observe that the total number of lists in levels 0 to L is at most $2^{L+1} < 2n$. Thus, applying a union bound over all lists yields the desired result. □

The preceding discussions focused on properties of the levels and their lists. Next, we will look at properties that hold for arbitrary individual nodes in a family tree. Let X be a node in a family tree. Both X.FIRSTCHILD and X.LEVELNEXT.FIRSTCHILD point to nodes in the same list (at level X.LEVEL $-$ 1). Consider the number of nodes in this list whose name IDs fall between X.NAMEID and X.LEVELNEXT.NAMEID. This is an important concept, so we give it a name.

Definition 4.3.1. Let X be a node. Let $\mathcal{L}$ be the level list containing X.FIRSTCHILD. We denote by CHILDREN(X) the sublist of $\mathcal{L}$ containing all nodes Y such that X.NAMEID $<$ Y.NAMEID and Y.NAMEID $<$ X.LEVELNEXT.NAMEID.

Intuitively, we would expect every node to have about two children since the lower level list is about twice as long. The following theorem shows that the expected length is indeed a constant.

Theorem 4.3.4. *For any node* X, |CHILDREN(X)| *is $O(1)$ in expectation and $O(\lg n)$ with high probability.*

Proof. Consider the process of traversing the name ID list, starting at X, and choosing a numeric ID and level for each of the nodes. Let $\mathcal{C}$ denote the list at level X.LEVEL-1 containing X's children. Our goal is to count the number of nodes that are chosen to belong to $\mathcal{C}$ before choosing X's successor in its level list. This process is a sequence of trials with three outcomes: on a "success", we have chosen X's successor; on a

"failure", we have added a new node to $\mathcal{C}$; otherwise, we have a "retry", indicating an unrelated node. Equivalently, we can eliminate the "retry" outcome by thinking of each trial as continuing until the first success or failure. Since there are now only two possible outcomes, it is a Bernoulli trial. Call a success in the Bernoulli trial a "heads" and a failure in the Bernoulli trial a "tails". The remainder of the proof bounds the number of tails before the first heads.

First, we must bound the probabilities of success and failure in the three-outcome trial. Let $k = \text{X.LEVEL} - 1$. For the probability of success (finding X's successor in its level list), we have

$$\begin{aligned} \Pr(\text{success}) &\geq 1/2^{k+1} L_{\max} = 1/2^{k+1}(c+2)\lg n \\ \Pr(\text{success}) &\leq 1/2^{k+1} L_{\min} = 1/2^{k+1}\lg(n/(c+2)\ln n) \end{aligned}$$

For the probability of failure (finding another node in $\mathcal{C}$), we have

$$\begin{aligned} \Pr(\text{failure}) &\geq 1/2^{k} L_{\max} = 1/2^{k}(c+2)\lg n \\ \Pr(\text{failure}) &\leq 1/2^{k} L_{\min} = 1/2^{k}\lg(n/(c+2)\ln n) \end{aligned}$$

Using these, we can bound the probability of a tails. Let $\Pr(\text{failure}) = \alpha/2^k \lg n$. Then we have

$$\begin{aligned} \Pr(\text{tails}) &= \Pr(\text{failure})/(\Pr(\text{failure}) + \Pr(\text{success})) \\ &< (\alpha/2^k \lg n)/(\alpha/2^k \lg n + 1/2^{k+1}(c+2)\lg n) \\ &= \alpha/(\alpha + 1/2(c+2)) \\ &= 1/(1 + 1/2(c+2)\alpha) \end{aligned}$$

To get an upper bound for Pr(tails), we apply an upper bound for α. By definition of α, we have $\alpha < \lg n/\lg(n/(c+2)\ln n)$. For sufficiently large n, we can bound $\alpha < 2$. Thus, for sufficiently large n, we can bound Pr(tails) by some constant $p < 1$. Then, the expected number of tails before the first heads is less than $p/(1-p) = O(1)$ as

this follows a negative binomial distribution. This shows that the expected number of children is $O(1)$.

To complete the proof, we compute a high probability bound on the number of children. The probability that we see at least k tails before a heads is p^k. If we let $\beta = 1/p$, then picking $k = c \log_\beta 2 \lg n$ gives a probability of $1/\beta^{c \log_\beta 2 \lg n} = 1/n^c$. □

Having defined the children of a node, we now define the parents.

Definition 4.3.2. For any node X, let PARENTS(X) = {Y | X ∈ CHILDREN(Y)}.

A slight variation on the proof of Theorem 4.3.4 yields the following result.

Theorem 4.3.5. *For any node* X, |PARENTS(X)| *is* $O(1)$ *in expectation and* $O(\lg n)$ *with high probability.*

The previous theorems examined the level lists. We now focus on the name ID list and the numeric ID list, both of which contain every node. The next theorem shows that no matter where we are in the name ID or numeric ID list, there is always a nearby node that is at a given level (provided that level is not too large).

Theorem 4.3.6. *The distance in the name ID list (or numeric ID list) from a node* X *to the nearest node at level* $L \in \{0, \ldots, L_{min} - 1\}$ *is* $O(\lg n)$ *in expectation and* $O(\lg^2 n)$ *with high probability.*

Proof. As in the proof of Theorem 4.3.4, imagine traversing through the name ID list (or numeric ID list) and choosing the level of each node as we encounter it. The expected distance to a node at level L is

$$\begin{aligned} \text{E[distance]} &< \textstyle\sum_{i=1}^{\infty} i(1 - 1/L_{\max})^{i-1}(1/L_{\min}) \\ &= (1/L_{\min}) \textstyle\sum_{i=1}^{\infty} i(1 - 1/L_{\max})^{i-1} \\ &= (1/L_{\min})/(1 - (1 - 1/L_{\max}))^2 \\ &= L_{\max}^2/L_{\min} \\ &= O(\lg n) \end{aligned}$$

The probability that there are no level i nodes within a distance of $kL_{\max}$ is less than $(1 - 1/L_{\max})^{kL_{\max}} < e^{-k}$. If we choose $k = c \ln n$, then there are no level i nodes within a distance of $c(c+2) \lg n \ln n = O(\lg^2 n)$ with probability less than $1/n^c$. □

Another variation on the proof of Theorem 4.3.4 yields the following result, which counts the number of "descendants" of a given node.

Theorem 4.3.7. *Let* X *be a node at level* L. *The number of nodes whose name ID is between* X.NAMEID *and* X.LEVELNEXT.NAMEID *is* $O(2^L \lg n)$ *in expectation and* $O(2^L \lg^2 n)$ *with high probability.*

4.4 Lookup Operations

In this section, we will describe the lookup operations and analyze their performance. In the next section, we will consider update operations.

The LOOKUP-BY-NAME-ID operation searches from a start node to find the node whose NAMEID is closest but less than or equal to a given destination name. Figure 4-2 contains the pseudocode for this operation. For the sake of simplicity, we have assumed that the destination name is greater than the start node's name. The other case is similar.

While similar in spirit to the LOOKUP-BY-NAME-ID algorithm of SkipNet, the Family Tree version must deal with much added complication. As in SkipNet, the plan is to start at a high level and move downward. Each time we move down a level, we further narrow in on the position of the destination.

In SkipNet, every node is at both high and low levels, so we can begin moving downward immediately from the highest level in the start node. In Family Trees, however, the start node may not be at a high level. Or the start node may be at too high of a level. Even if we move immediately down to the child node, we may advance far past the destination. We solve both problems by first searching through the name ID list for a nearby level 0 node. This is accomplished by the call to LINEAR-SEARCH-FOR-LEVEL-0.

```
LOOKUP-BY-NAME-ID( start , dest )
 37  X ← LINEAR-SEARCH-FOR-LEVEL-0( start , dest )
 38  X ← FIND-CLOSEST-AT-LEVEL-0( X , dest )
 39  X ← LINEAR-SEARCH-FOR-DESTINATION( X , dest )
 40  return X

LINEAR-SEARCH-FOR-LEVEL-0( X , dest )
 41  while X.NAMENEXT ≠ NIL and
 42        X.NAMENEXT.NAMEID < dest and
 43        X.LEVEL > 0
 44   do X ← X.NAMENEXT
 45  return X

FIND-CLOSEST-AT-LEVEL-0( X , dest )
 46  left ← X.NAMEID
 47  while true
 48   do if X.LEVELNEXT = NIL or
 49         dest < X.LEVELNEXT.NAMEID
 50         then break
 51      P ← X.MOTHER or X.FATHER at random
 52      if P ≠ NIL
 53         then X ← P
 54         else break
 55      X ← LEVEL-SEARCH-BY-NAME-ID ( X , left )
 56  while true
 57   do if X.LEVEL > 0
 58         then X ← X.FIRSTCHILD
 59         else break
 60      X ← LEVEL-SEARCH-BY-NAME-ID ( X , dest )
 61  return X

LEVEL-SEARCH-BY-NAME-ID( X , name )
 62  while X.LEVELNEXT ≠ NIL and
 63        X.LEVELNEXT.NAMEID < name
 64   do X ← X.LEVELNEXT
 65  while X.LEVELPREV ≠ NIL and
 66        name < X.NAMEID
 67   do X ← X.LEVELPREV
 68  return X

LINEAR-SEARCH-FOR-DESTINATION( X , dest )
 69  while X.NAMENEXT ≠ NIL and
 70        X.NAMENEXT.NAMEID < dest
 71   do X ← X.NAMENEXT
 72  return X
```

Figure 4-2: LOOK-BY-NAME-ID *finds the node whose name ID is closest to the given destination.*

Once we have found a level 0 node, we use parent pointers to move up to a high level. We continue to advance until we get to a level high enough that the next node in that level list is beyond the destination. This is where the SkipNet lookup would start. From here, we can move down levels in a manner analogous to SkipNet. Both the upward and downward parts are performed by FIND-CLOSEST-AT-LEVEL-0. This routine starts at the closest level 0 node to the start node and finishes at the closest level 0 node to the destination. Each time we move up or down a level, we call LEVEL-SEARCH-BY-NAME-ID to make sure we stay at the node closest to the name in question. On the way up, we stay at the node closest to the name ID of the level 0 node at which we started. On the way down, we stay at the node closest to the lookup value. The lookup switches from upward to downward movement when we reach a high enough level that the closest node to the start name ID is also closest to the lookup value. (At the highest level, the list would contain only a single node, so this would be satisfied trivially.)

In SkipNet, the lookup ends when we reach the bottom level since every node is in the level 0 list. However, in Family Trees, this list contains only the level 0 nodes. We find the node that is closest overall by again searching through the name ID list, which contains all nodes. This search is implemented in LINEAR-SEARCH-FOR-DESTINATION. The correctness of the algorithm is assured by this last linear search. Even if FIND-CLOSEST-AT-LEVEL-0 left us far from the destination, this linear search will make sure we return the correct node.

Next, let's analyze the performance of this algorithm. Theorem 4.3.6 shows that LINEAR-SEARCH-FOR-LEVEL-0 requires $O(\lg n)$ hops in expectation and $O(\lg^2 n)$ hops with high probability. Since the number of levels is $O(\lg n)$ with high probability, the two loops in FIND-CLOSEST-AT-LEVEL-0 will execute no more than $O(\lg n)$ times. Each iteration of the loops makes $O(1)$ hops except for the call to LEVEL-SEARCH-BY-NAME-ID. Each node traversed in this call is a parent of the sucessor of the node reached by the previous iteration. Thus, the total number of hops is $O(1)$ in expectation and $O(\lg n)$ with high probability by Theorem 4.3.5, and the total number of hops for LEVEL-SEARCH-BY-NAME-ID is $O(\lg n)$ in expectation and $O(\lg^2 n)$

with high probability.[5] Lastly, the analysis of LINEAR-SEARCH-FOR-DESTINATION is identical to that of LINEAR-SEARCH-FOR-LEVEL-0 since the number of hops is just the distance from the destination to the closest node at level 0. Thus, we have shown that LOOKUP-BY-NAME-ID requires $O(\lg n)$ in expectation and $O(\lg^2 n)$ hops with high probability.

Note that the pseudocode presented in Figure 4-2 is sequential. A distributed implementation would execute various portions of that code at different nodes. As a practical matter, distributed nodes would cache the name IDs of their neighboring nodes. Using this cache, the distributed search algorithm can determine that a node is beyond the destination without accessing it. We will assume that such caching occurs for the analysis below.

Next, we will show that Family Tree name ID lookups have good locality. First, recall that the algorithm begins by finding a nearby level 0 node. To see why this first phase is important for locality, suppose that the start node is at a high level and that the destination name is very close to that of start node. In this case, the start node's LEVELNEXT and FIRSTCHILD pointers are both likely to point well beyond the destination node, so traversing them would result in poor locality. Instead, we find a nearby node at level 0 where the expected distance to its successor in the level list is as small as possible.

The two linear searches performed in the name ID lookup algorithm have *strict* locality. They will never access any node whose name ID is not between those of the start and destination nodes. However, the upward part of the algorithm in FIND-CLOSEST-LEVEL-0, as written, could access a node outside this range when it follows a parent pointer. We could achieve strict locality here as well by adding a second mother and father pointer that point to the node whose name ID is closest but greater than the node in question. This would allow us to determine that we have reached the highest level necessary without stepping outside the range between the start and destination nodes. We would similarly need to add a child pointer that points to the

[5]There is slack in this part of the argument. It can be shown that LEVEL-SEARCH-BY-NAME-ID requires $O(\lg n)$ hops with high probability. However, since this would not improve the bounds for the algorithm overall, we omit the more complicated argument.

node whose name ID is closest but less than the node in question. So with three additional pointers, we could get strict locality.

However, even without these additional pointers, we have locality in a probabilistic sense. Specifically, we can show that the expected maximum distance that we will step outside the range from the start to destination nodes is $O(D)$, where D is the distance between the start and destination nodes in the name ID list. We will argue this for the upward part of the algorithm. The argument for the downward part is symmetric.

First, by Theorem 4.3.7, the expected distance from the leftmost node traversed at level j to *start* is $O(2^j \lg n)$. We can bound the expected maximum distance of all nodes traversed up through level j by the expected sum of these distances. Since these distances increase geometrically, this sum is also $O(2^j \lg n)$. Next, we can condition the expected maximum distance on the highest level reached, which will be j for the previous calculation. Define h to be the smallest value such that $D \leq 2^h \lg n$. Intuitively, we would expect that the highest level reached is h or higher. A short calculation that shows that the probability that the highest level reached is $h + i$ is at most $1/2^{i(i-1)/2}$. Since these probabilities decrease exponentially faster than the conditioned maximum distances increase, the expected maximum distance is $O(D)$.

Next, we turn to lookups by numeric ID. Again, the algorithm is similar in spirit to that of SkipNet but with added complication. The pseudocode of this operation, shown in Figure 4-3, is shorter than that of name ID lookup because it reuses several of those routines.

As with name ID lookups, we must first find a level 0 node. This is accomplished as before by a call to LINEAR-SEARCH-FOR-LEVEL-0. Then, we move upward according to the bits of the lookup value. As with name ID lookup, a SkipNet lookup complete once we reach the top-most list, whereas in Family Trees, the top-most list only contains the closest node in that level, not the closest node overall. To find the true destination, we perform a linear search in the numeric ID list by calling LINEAR-SEARCH-BY-NUM-ID.

The pseudocode adds a bit more complication by allowing finite sequences of bits

```
LOOKUP-BY-NUM-ID( start , bits )
 73  name ← start . NAMEID
 74  X ← LINEAR-SEARCH-FOR-LEVEL-0( start , name )
 75  while true
 76   do if X.LEVEL = | bits |
 77         then return X
 78      if bits [ X.LEVEL ] = 0
 79         then P ← X.MOTHER
 80         else P ← X.FATHER
 81      if P = NIL
 82         then break
 83       X ← LEVEL-SEARCH-BY-NAME-ID( P , name )
 84  X ← LINEAR-SEARCH-BY-NUM-ID( X , bits , | bits |)
 85  if | bits | < ∞
 86    then X ← LEVEL-SEARCH-BY-NAME-ID( X , name )
 87  return X

LEVEL-SEARCH-BY-NUM-ID( X , value )
 88  while X.LEVELNEXT ≠ NIL and
 89        X.LEVELNEXT.NUMID < value
 90   do X ← X.LEVELNEXT
 91  while X.LEVELPREV ≠ NIL and
 92        value < X.NUMID
 93   do X ← X.LEVELPREV
 94  return X
```

Figure 4-3: *The* LOOKUP-BY-NUM-ID *function finds the node whose numeric ID is closest to the given value.*

to be looked up. Any real input can always be considered to be infinite by appending zeros. These finite sequences are used in the case where we want to find the closest node in a particular level. The level is given by the length of *bits*. We can see that the pseudocode will terminate early in this case. This behavior will be used for the update operations discussed in the next section.

Lastly, let's analyze the performance of this algorithm. The last subsection showed that LINEAR-SEARCH-FOR-LEVEL-0 requires $O(\lg n)$ hops in expectation and $O(\lg^2 n)$ with high probability. The upward part of this algorithm is identical to that of name ID lookup (except that the parent choices are given), so the same time bounds apply. The performance of the last part of the algorithm, LINEAR-SEARCH-BY-NUM-ID, depends on the level reached in the second phase. By Theorem 4.3.3, we will reach *at least* level $k = \lg n - 2\lg\lg n - 2\lg(c+2)$ with high probability. This leaves a range of size at most $1/2^k = \lg^2 n \cdot \lg^2(c+2)/n$ in numeric ID space to be searched, so the expected number of nodes we will traverse in LINEAR-SEARCH-BY-NUM-ID is $O(\lg^2 n)$. Since this is $\Omega(\lg n)$, the Chernoff bound implies that the number of nodes we will actually search within a constant factor of its expected length with high probability. Putting this all together, we have shown that numeric ID lookup requires $O(\lg^2 n)$ hops with high probability.

To get a bound on the expected number of hops in LINEAR-SEARCH-BY-NUM-ID, we need to do a little more work. First, notice that if we get to level $L_{\min}$, the expected number of nodes we will search is $(c+2)\ln n$. For higher levels, the expected number of nodes can only be smaller. Thus, if we get to a level this high, the expected number of hops is $O(\lg n)$. So now, our only worry is for levels between $\lg n - 2\lg\lg n - 2\lg(c+2)$ and $L_{\min} = \lg n - \lg\ln n - \lg(c+2)$. We can compute the expected value in these cases by conditioning on the last level we reach. We will

write each such level as $L_{\min} - k$, where k ranges from 0 to $\lg[(c+2)\lg n]$.[6]

$$\mathrm{E}[\text{hops}] = \sum_{k=0}^{\lg[(c+2)\lg n]} \Pr\left(L_{\min}+k \text{ is last level}\right) \mathrm{E}\left[\text{hops} \mid L_{\min}+k \text{ is last level}\right]$$

Next, we can use the fact that the probability that the last level is L, which is the probability that none of the levels up to L are empty but level $L+1$ is, is no more than the probability that level $L+1$ is empty. Thus, we can see that the expected number of hops is

$$\begin{aligned}
\mathrm{E}[\text{hops}] &\leq \sum_{k=0}^{\lg[(c+2)\lg n]} \left(1 - \frac{1}{2^{L_{\min}-k+1}L_{\max}}\right)^n \frac{n}{2^{L_{\min}-k}} \\
&= \sum_{k=0}^{\lg[(c+2)\lg n]} \left(1 - \frac{2^k(\ln 2/2)}{n}\right)^n 2^k(c+2)\ln n \\
&\leq \sum_{t=2^k=1}^{(c+2)\lg n} \left(1 - \frac{t(\ln 2/2)}{n}\right)^n t(c+2)\ln n \\
&< \sum_{t=1}^{(c+2)\lg n} \exp(-(\ln 2/2)t)t(c+2)\ln n \\
&= O(\lg n)
\end{aligned}$$

This last part follows since $\int_1^\infty te^{-t}$ integrates to a constant. Thus, we have shown that LOOKUP-BY-NUM-ID requires $O(\lg n)$ time in expectation and $O(\lg^2 n)$ time with high probability.

As we saw in chapter two, SkipNet also supports name-constrained numeric ID lookups, which as we saw in chapter three, is a very useful operation. It is not at all clear that Family Trees can support this operation. We will address this issue in detail in chapter five after we have discussed modifications to these algorithms to improve their performance in practice.

[6]We should take the floors of these numbers to make sure they are integers. However, this only affects lower order terms that disappear in the O-notation. So we will ignore these sorts of issues for clarity in this argument.

4.4.1 Congestion

Another important property of SkipNet lookups is that they spread the load of requests evenly across the network, assuming that the requests themselves are spread evenly. This is in contrast to, say, a balanced binary tree, where half of all paths from one node to another go through the root node. In this section, we will make these notions precise and prove that Family Trees share the same property.

We define the *congestion at node* X to be the probability that a search operation with source S and target T, chosen uniformly at random, traverses X. For example, the congestion at the root node of a balanced binary tree is at least $(\frac{n}{2})^2/\binom{n}{2} = \Theta(1)$. Congestion of $\Theta(\lg n/n)$ is optimal when the nodes have constant degree because $\Theta(\lg n)$ nodes must be traversed in most search paths. The theorem below shows that Family Trees achieve the optimal bound.

This definition of congestion at a node is a probability where the unknown random variables are the numeric IDs and levels of all nodes in the data structure, the random bits used in the search itself, and the choice of S and T. If we imagine exposing the random bits used by every node in the data structure, then we could look at the *congestion of the Family Tree*, which is defined to be the maximum congestion at any node. (Note that this is a probability on the unexposed random variables.) The maximum congestion at any node in the network is important theoretically and in practice. The following theorem also shows that the congestion of a Family Tree does not deviate much from the congestion at an arbitrary node.

Theorem 4.4.1. *The congestion at any particular node in a Family Tree is* $O(\lg n/n)$. *The congestion of the Family Tree is* $O(\lg^2 n/n)$ *with high probability.*

Proof. We will analyze LOOKUP-BY-NAME-ID. The analysis for LOOKUP-BY-NUM-ID is nearly identical. Let X be any node in the family tree. X could be traversed during any of the four parts of the search algorithm: (1) linear search for a level 0 node, (2) moving up to a high level, (3) moving down to the closest level 0 node, and (4) linear search for the destination. We will consider each in turn and show that, in each, the probability that X is traversed is $O(\lg n/n)$ in expectation and $O(\lg^2 n/n)$

with high probability[7].

To be traversed in the first part, X must lie between S and the closest level 0 node to the right of S. Theorem 4.3.6 showed that this distance is $O(\lg n)$ in expectation and $O(\lg^2 n)$ with high probability. Thus, the probability that X lies between a randomly chosen S and its closest level 0 node is $O(\lg n/n)$ in expectation and $O(\lg^2 n/n)$ with high probability.

For the node X to be traversed in the second part (upward movement), both of the following conditions must hold. First, the X.LEVEL random parent choices must match the corresponding bits of X's numeric ID. This occurs with probability $1/2^{\text{X.LEVEL}}$. Second, S must be between X and X.LEVELNEXT.FIRSTCHILD. Since all nodes traversed in the upward search are before S, we will not traverse X if S is before it.[8] If S is after X.LEVELNEXT.FIRSTCHILD, then the closest node to S in the previous level is to the right of X.LEVELNEXT, so the parent of that node will be X.LEVELNEXT or further to the right. We can use two applications of Theorem 4.3.7 to bound the number of nodes between X and X.LEVELNEXT.FIRSTCHILD as $O(2^{\text{X.LEVEL}} \lg n)$. Thus, the probability that a randomly chosen S is in this set is $O(2^{\text{X.LEVEL}} \lg n/n)$ in expectation and $O(2^{\text{X.LEVEL}} \lg^2 n/n)$ with high probability. Since these two conditions are independent, we can multiply them to show that the probability that X is traversed in the second phase is $O(\lg n/n)$ in expectation and $O(\lg^2 n/n)$ with high probability.

The analysis of the third part (downward movement) is symmetric to that of the second, and the analysis of the fourth part is identical to that of the first. Adding together the congestion due to each phase, we obtain a bound of $O(\lg n/n)$ in expectation and $O(\lg^2 n/n)$ with high probability. □

[7]By "in expectation" and "with high probability", we are referring to the outcome when the structure of the Family Tree is revealed, which is still a probability over the randomness used in the search and the choice of start and destination. The expected value of this probability is identical to the probability when no information is known, which is our definition of congestion at the node.

[8]We ignore the case where S is before X but the closest level-0 node is after X. This is counterbalanced by the analogous case where S is between X and X.LEVELNEXT.FIRSTCHILD but the closest level-0 node is not, which we do include in the probability.

4.5 Update Operations

In this section, we describe the insert and delete operations and analyze their performance. Analogous congestion bounds follow immediately from the definitions of the operations.

Pseudocode for joining a new node into the network is shown in Figure 4-4. Most of the work required is accomplished by calls to the search operations described in section 4.4. Finding the predecessor of the new node in the name ID list is a simple matter of calling LOOKUP-BY-NAME-ID. Once this predecessor has been found, linking the node into this doubly-linked list just requires updating four pointers. Inserting the new node into the numeric ID list is identical except that the call is instead made to LOOKUP-BY-NUM-ID. These two parts are accomplished by the calls to. After this, it remains to set the level list pointers and the inter-level pointers.

In order to choose a level for the new node X, we must first compute the estimate $\ell \approx \lfloor \lg n \rfloor$. To do this, we subtract the numeric IDs of X and its successor (mod 1) and find the first non-zero bit. This will be found before the $(c \lg n)$-th bit with high probability. Next, we choose X.LEVEL uniformly at random from $\{0, \ldots, \ell - 1\}$. Once X has a level, we perform a LOOKUP-BY-NUM-ID, using just the first X.LEVEL bits of X.NUMID, to find the predecessor of X in its level list. (See section 4.4 for more information on numeric ID lookups using finite seqeunces of bits.) Similarly, we can find X.FIRSTCHILD by performing a LOOKUP-BY-NUM-ID using just the first X.LEVEL $-$ 1 bits of the numeric ID. We then enumerate all the children of X and update their appropriate parent pointers to point to X. We handle X.MOTHER and X.FATHER similarly. It is worth noting that most of these calls to LOOKUP-BY-NUM-ID will be retracing the steps of other calls. It would be a simple matter to remove this in a real implementation. However, we have written each call as separate to simplify both the pseudocode and its analysis.

Lastly, we turn to the predecessor of X in the numeric ID list. As mentioned in the proof of Corollary 4.3.1, we must allow this node to re-estimate n and choose a new level. This ensures that the shape of the Family Tree is independent of history.

```
JOIN( X )
 95  pred = LOOKUP-BY-NAME-ID( X , X.NAMEID )
 96  ADD-TO-NAME-ID-LIST( X , pred )
 97  pred = LOOKUP-BY-NUM-ID( X , X.NUMID )
 98  ADD-TO-NUM-ID-LIST( X , pred )
 99  ADD-TO-LEVEL( X )
100  REMOVE-FROM-LEVEL( pred )
101  ADD-TO-LEVEL( pred )

ADD-TO-LEVEL( X )
102  X.LEVEL ← RANDOM(0, ESTIMATE-N( X ) − 1)
103  pred ← LOOKUP-BY-NUM-ID( X , X.NUMID [1 ... X.LEVEL ])
104  ADD-TO-LEVEL-LIST( X , pred )
105  X.MOTHER ← LOOKUP-BY-NUM-ID( X , X.NUMID [1 ... X.LEVEL ] + [0])
106  X.FATHER ← LOOKUP-BY-NUM-ID( X , X.NUMID [1 ... X.LEVEL ] + [1])
107  ADD-CHILD( X.MOTHER , X )
108  ADD-CHILD( X.FATHER , X )
109  X.FIRSTCHILD ← LOOKUP-BY-NUM-ID( X , X.NUMID [1 ... X.LEVEL − 1])
110  ADD-PARENT( X.FIRSTCHILD , X )

ADD-CHILD( X , child )
111  if X.NAMEID < child.NAMEID < X.FIRSTCHILD.NAMEID
112    then X.FIRSTCHILD = child

ADD-PARENT( X , parent )
113  while X ≠ parent .LEVELNEXT.FIRSTCHILD
114   do if parent.NAMEID < X.NAMEID
115        then if X.MOTHER = parent.LEVELPREV
116               then X.MOTHER ← parent
117             if X.FATHER = parent.LEVELPREV
118               then X.FATHER ← parent
119      X ← X.LEVELNEXT

ESTIMATE-N( X )
120  before ← X.NUMID < X.NUMNEXT.NUMID
121  for i ← 1 to ∞
122  do b1 ← X.NUMID [i]
123     b2 ← X.NUMNEXT.NUMID [i]
124     if before
125       then b ← b2 − b1
126       else b ← b1 + 1 − b2
127     if b = 1
128       then return i
129     if b = 2
130       then return i − 1
```

Figure 4-4: JOIN adds a new node into the network. The three ADD-TO-*-LIST functions, which simply add a node into a linked list, have been omitted for the sake of brevity.

```
LEAVE( X )
131  REMOVE-FROM-NAME-ID-LIST( X )
132  REMOVE-FROM-NUM-ID-LIST( X )
133  REMOVE-FROM-LEVEL( X )
134  REMOVE-FROM-LEVEL( X.PREV )
135  ADD-TO-LEVEL( X.NUMPREV )

REMOVE-FROM-LEVEL( X )
136  REMOVE-FROM-LEVEL-LIST( X )
137  REMOVE-CHILD( X.MOTHER , X )
138  REMOVE-CHILD( X.FATHER , X )
139  REMOVE-PARENT( X.FIRSTCHILD , X )

REMOVE-CHILD( X , child )
140  if X.FIRSTCHILD = child
141    then X.FIRSTCHILD = child.LEVELNEXT

REMOVE-PARENT( X , parent )
142  while X ≠ parent .LEVELNEXT.FIRSTCHILD
143   do if X.MOTHER = parent
144        then X.MOTHER ← parent.LEVELPREV
145      if X.FATHER = parent
146        then X.FATHER ← parent.LEVELPREV
147      X ← X.LEVELNEXT
```

Figure 4-5: LEAVE removes a node from the network. The three REMOVE-FROM-*-LIST functions, which simply remove a node from a linked list, have been omitted for the sake of brevity.

Note that the predecessor's numeric ID does not change, only its level does, so the re-estimation process does not cascade to other nodes. The procedure for estimating n and choosing a level was described in the previous paragraph. The procedure for removing this node from its old level is described in the next subsection.

It is easy to see that the running time of JOIN is dominated by the six calls to search operations. We argued above that computing the estimate of $\lg n$ takes $O(\lg n)$ time with high probability. The only other work is updating the pointers. There are only nine outbound pointers. The number of inbound pointers is $O(\lg n)$ with high probability by Theorem 4.3.4 and Theorem 4.3.5. Thus, the total time required is $O(\lg n)$ in expectation and $O(\lg^2 n)$ with high probability.

The algorithm for separating a node from the network is even simpler. Pseudocode

is shown in Figure 4-5. No lookups are required for this operation. First, we remove the node from the name ID, numeric ID, and level lists. Removing from the level also requires updating any children and parents that may have been affected. Lastly, we must allow the predecessor of the removed node to re-estimate n and choose a new level, so that the structure of the network stays independent of history. As mentioned above, the re-estimate process does not cascade to other nodes.

The only significant work done by this algorithm is in iterating parents and children. As we have already seen, the number of such nodes is $O(1)$ in expectation and $O(\lg n)$ with high probability. Thus, this algorithm requires $O(1)$ hops in expectation and $O(\lg n)$ hops with high probability.

4.6 Conclusions

In this chapter, we have seen the design of Family Trees, the first ordered peer-to-peer overlay network that uses only $O(1)$ pointers per node. We have seen that Family Trees can perform name ID and numeric ID lookups. Name ID lookups have locality, which can be made strict with the addition of three more pointers. All of the operations (lookups and updates) are efficient, requiring only $O(\lg n)$ hops in expectation and $O(\lg^2 n)$ with high probability. And all of the operations have optimal congestion in expectation.

In the next section, we will look at the performance of Family Trees in a practical setting. In addition to confirming the theoretical work in this chapter, we will also look optimizations for practical use and look at the issue of performing name-constrained numeric ID lookups in Family Trees.

Chapter 5

Family Trees In Practice

In this chapter, we will look at results of experiments studying the performance of Family Trees on actual networks. We begin with an overview of the experimental setup. Second, we look at how to optimize Family Trees to reduce the total latency of lookups (rather than the number of hops) and show experimental results demonstrating their effectiveness. Third, we look at experiments that verify the theory of the previous chapter, specifically the asymptotic performance of name and numeric ID lookups in terms of hops per query and congestion in the network. Fourth, we look at whether Family Trees can implement name-constrained numeric ID lookups. We conclude in the last section.

5.1 Experimental Setup

For our experiments, we implemented the Family Tree and SkipNet protocols on top of a network simulator. We used the `p2psim` network simulator [9] developed at MIT. This allowed us to run, on a single machine, experiments that simulated large numbers of nodes connected by a vast network.

A central concern in any simulated network is the model used to generate the network. Previous simulations have used models such Transit-Stub [28], Mercator [25] and King [11]. It is important to note, however, that these models differ only in the *latencies* seen in the network. It is assumed that the underlying network is

connected (any node can send to any other), so only the delay in sending can vary from one model to another.[1] Only the results of section 5.2 are dependent on these latencies. Thus, the results of every other section are unaffected by the differences between these models.

Unlike simulations of unordered networks, however, we also require that our network model generate friendly names for each node. Since none of the models mentioned above generates these names, we had to do so ourselves. However, these models are not all equal with respect to how well they facilitate the generation of names. For our experiments, we choose the Transit-Stub model because it provides the most help for generating names.

The Transit-Stub model directly simulates the hierarchy of the Internet. It starts with a random graph, which models the interconnections between transit networks in the Internet. Each of these nodes is then replaced with a random graph, which becomes a transit network. Each node in a transit network is then connected to some number of stub domains, which are also random graphs. Each stub domain is intended to model the network of an organization (such as a university or company), with nodes being its routers. The individual machines of that organization would be connected to the router nodes.

Transit-Stub does not model the hierarchy within an organization. The network of a large organization most likely does not look like a random graph. A university, for example, probably has networks for each building or department, which are connected together by a university-wide transit network. The network within a department may be separated into floors or groups.

Our intention is to model the hierarchy that shows through in domain names, which we use as name IDs. Realistically, we can expect to see at most four levels of hierarchy in practice, with levels corresponding to organizations, departments, groups, and machines. In order to generate names of this form, we make one change to our interpretation of the Transit-Stub models. These models only include a single level of

[1] If the network models the bandwidth between connections, then that too could vary. However, we ignored bandwidth constraints, which were almost certainly not violated, in our study.

hierarchy in the stub domains (hosts are underneath routers). To get a second level, we interpret all stubs attached to the same transit node as being part of the same organization, with each individual stub domain corresponding to a department. In this manner, we can generate name IDs for each node with up to four parts: transit node (organization), stub domain (department), stub node (group), and machine. If only a single stub domain is connected to a transit node, then names within that organization will have only three levels of hierarchy.

It is clear that name IDs will better correlate to latencies when all four levels of hierarchy are present. However, it is important to note that this correlation will not degrade much when only three levels are present. This is because the level that is most likely missing is the group level. If this level is removed from the domain names, then the names will not correlate well with distances between members of the same department. However, these distances are very small compared to the distances between departments and between organizations. Thus, removing this level reduces the correlation of names and distances by only a small amount, so general performance, when measured in terms of latency, will likewise degrade by at most a small amount.

In our experiments, we used the `ts100` sample model provided with the Transit-Stub model generators. This model includes only a single transit domain. Since the hierarchy of the transit domain is not reflected in domain names, using a single random graph for the transit layer is a worst case for us. The model includes (on average) four organizations. Each organization has (on average) three departments. (Since there are four organizations, often one of them will have have only a single department and, hence, only three levels of hierarchy.) Each department includes (on average) four groups. This gives a total of 96 stub nodes (routers). We assign 1000 hosts to these routers uniformly at random.

Each result presented below is averaged over 40 trials with those parameters. The `ts100` sample model includes 10 instances. We ran each experiment four times against each instance (for a total of 40 trials) in order to average over the randomness that is present in each trial (even with the same parameters).

Some other simulations (such as [12]) have been run on networks with as many as $100,000$ nodes. In our case, the choice of 1000 nodes was due mostly to the speed of the simulator and the large number of trials (40) that we wanted to average the results over. However, it is important to note that the advantage of Family Trees grows as n gets larger. The gap between $2 \lg n$ and 9 (the numbers of pointers in SkipNet and Family Trees, respectively) grows as n gets larger. Furthermore, the high probability bounds that we proved in the last chapter hold with higher probability as n gets larger. Thus, 1000 nodes is more of a "worst-case" for Family Trees than the $100,000$ node networks that some others have tried, so it made a good choice for our study.

5.2 Optimizing for Latency

Experimental evaluations of peer to peer networks typically focus on latency as the most important criterion. The number of hops per request is also very important because (1) each hop directly consumes network and CPU resources and (2) it gives a (weak) bound on the lookup latency by multiplying the number of hops by the maximum point-to-point latency. However, it seems quite possible that latency will become a problem for practical applications long before network bandwidth and CPU resources. Furthermore, a latency of number of hops times the maximum point-to-point latency would almost certainly not be acceptable in a real application, given that each hop could take over a second to deliver across the Internet. Thus, if Family Trees are to see practical use, we must show that their latency per request compares favorably to those of other networks.

In this section, we will look at what changes are necessary in order to achieve good performance in terms of latency. We will focus on numeric ID lookups for two reasons. First, since unordered networks cannot perform name ID lookups, there is less need for comparison. Second, all of the techniques we optimizations we describe can also be applied to name ID lookups.

Recall that the results of this section are (the only results) dependent on the model used to generate the network. As described above, we will use the Transit-Stub

model. However, similar experiments of SkipNet [12] tried both the Transit-Stub and the Mercator models and found similar results on both. Given the similar structure of Family Trees and SkipNet, we would expect the results of this chapter to apply to the Mercator model as well.

Next, we will begin describing our optimizations for latency. Recall that the numeric ID lookup algorithm consists of three parts. In the first part, we search for a nearby level-0 node. In the second part, we move up levels trying to match as many bits of the query ID as possible. In the third part, we search through the numeric ID list to find the node that is truly closest. We will discuss each part in turn to see how it can be optimized for latency.

The first part of numeric ID lookup, searching for a level-0 node, can be eliminated very easily. We simply cache a pointer to this node. Every lookup (both numeric and name ID) will most likely go through this node, so we can save $O(\lg n)$ hops and the associated latencies by jumping directly to that node. To keep the pointer up to date, we have two options: we could refresh the pointer every so-many requests or we could require any new level-0 node to notify every node between it and the previous level-0 node in the name ID list. Either of these options changes the performance of the lookup, join, and leave algorithms by only a constant factor (in terms of hops). Thus, we can see that the first part of the lookup is easy to eliminate.

The second part of numeric ID lookup, moving upward through the levels, cannot be eliminated. Essentially, the same process is performed by SkipNet, Pastry, and Tapestry. However, these other networks have an advantage over Family Trees. Each of their nodes can process any request: each node can route a request regardless of the size of the prefix match between the query bits and the current node's numeric ID. In Family Trees, however, each node can only route requests at its level, so it can only route a request whose prefix match is exactly LEVEL bits. This gives SkipNet and the other networks a boost in performance because, each time the request is forwarded, the size of the prefix match will increase by at least 2 with probability $\frac{1}{2}$. This decreases the number of expected hops by a constant factor, which also translates into an improvement in latency. To make Family Trees competitive, we must find a

way to decrease our hops in the second part of the algorithm by a constant factor as well. We will look for a way to decrease hops by adding more pointers.

We have now seen two instances where additional pointers are used to decrease hops and latencies: Family Tree nodes caching the nearest level-0 node and SkipNet nodes being in multiple level lists. However, it is generally true that *we can decrease hops and latencies by adding more pointers.* We will now look at a more general ways to trade off space (pointers) for time (latency) in Family Trees and SkipNet.

In Family Trees, we have one pointer to a parent node with next numeric ID bit 0 (mother) and one pointer to a parent with next bit 1 (father). In SkipNet, one of these parent nodes will be the node itself and the other we find by looking through nearby nodes in the level list. It would be useful in SkipNet to cache the pointer to the other parent since the hops required to find it would otherwise be performed on every lookup.[2] However, instead of maintaining just 1 pointer for each parent, we could maintain k pointers to parents of each gender (mother and father) that are close to the current node ($2k$ pointers all together).

How will these pointers help us? First of all, when we need to forward a request to a parent, we can send it to the closest of the k nodes we found. This reduces latency while still ensuring that each hop matches another bit of the query ID, so we are still guaranteed correctness and $O(\lg n)$ hops. Of course, we could find the closest k during JOIN but only record the truly closest node. We need to maintain all k, though, in order to help other nodes efficiently find their own closest k. The algorithm for doing this comes from Pastry [3]. We start out by finding a single level-0 node very close to ourselves. This node will have pointers to k close level-1 nodes with first numeric ID bit 0 and k close level-1 nodes with first bit 1. If we are a level-0 node, then these are the pointers we need. If not, we take the k that match the first bit of our numeric ID, contact the closest of these, and construct a list of $2k$ level-2 nodes: k with bit prefix $x0$ and k with bit prefix $x1$, where x is the first bit of our numeric ID. If we are a level-1 node, then these are our pointers. Otherwise, we continue on like this until we reach our level. The key to the effectiveness of this heuristic is that, when we get

[2]This is yet another instance of decreasing latency by adding pointers.

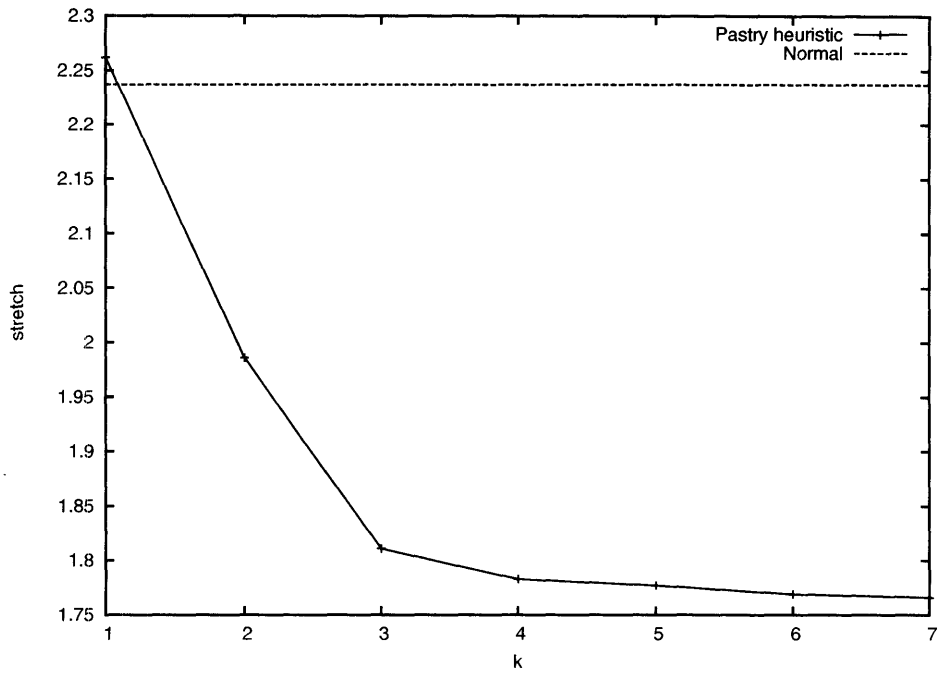

Figure 5-1: *Shows the relationship between the number of pointers (k) used in the Pastry heuristic and the average stretch of a lookup. The other line shows stretch with normal SkipNet pointers but caching the other parent.*

k nodes that are close to a node that is close to us, one of those k nodes is likely to be close to us as well. This can be proven formally if the triangle inequality holds. It doesn't in practice, but this heuristic is still effective.

Thus, we have a seen our first general way to trade off space for time: by increasing k, we maintain more pointers, but we will decrease the latency of each request. Figure 5-1 shows the effect of increasing k on the performance of lookups in a SkipNet network.[3] The vertical axis is not latency but relative delay penalty or *stretch*, which is the actual latency of routing the lookup divided by the true latency between the source and destination. (This measure actually overestimates the penalty because it does not include the time for the response message, which can be sent directly to the source. A stretch of $1+\delta$ would be $1+\frac{1}{2}\delta$ if we included the response. It is important to note that this transformation would not affect our comparisons of different algorithms.) We can see in Figure 5-1 that increasing k continues to decrease the stretch,

[3]As noted above, the results would be equivalent for Pastry and Tapastry since the numeric ID lookup algorithms are the same.

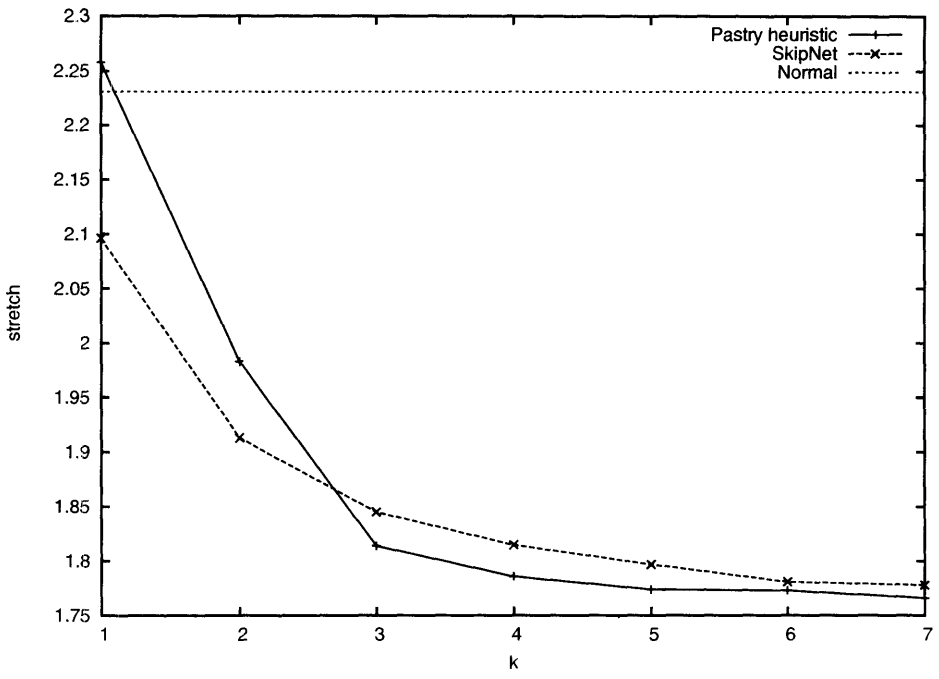

Figure 5-2: *Shows the effectiveness of sampling from the name ID list and the Pastry heuristic as functions of k.*

although there are diminishing returns.

The other line in Figure 5-1 shows the stretch when we just cache the parent that we would find using the normal SkipNet lookup algorithm, that is, the parent whose name ID is closest to that of the child. The figure shows that this pointer outperforms a single pointer generated using the Pastry heuristic. This is true even though we are allowing the Pastry heuristic to cheat: at level 0, they are finding the *true closest* k nodes, even though this is impossible to achieve exactly in practice.[4] Thus, we can see that, if we are going to cache only a single pointer, we are best off just caching the normal SkipNet parent and not implementing the Pastry heuristic at all.

This fact also demonstrates that name IDs correlate well (though not perfectly) with latency. We know from Figure 5-1 that sampling more nodes (larger k) allows us to find nodes that are closer to us and decrease stretch. However, the node that

[4]In the experiment, we are finding them by checking against every other node. This would be too slow in practice, so some additional heuristic would have to be used to find this node. The second heuristic would certainly degrade the performance of these pointers. Though, it would remain true that increasing k decreases stretch.

is closest amongst these k is likely to be close in name ID. This suggests another approach to finding the closest parent: simply look at the closest k nodes by name ID in the parent list. Figure 5-2 shows effectiveness of this approach as we increase k. We can see that this approach is just as effective as the Pastry heuristic. It should be mentioned that, in the experiment, we are actually sampling k nodes in each direction in the name ID list, for a total of $2k$ nodes. However, we have afforded the Pastry heuristic even more: it is generating the k closest nodes at the next level by retrieving from $2k$ next-level nodes from *each of the k nodes*. Thus, the Pastry heuristic is sampling roughly $2k^2$ pointers. Given this, we can see that sampling the name ID list is much more efficient while being just as effective.

In fact, sampling in the name ID list has an even more important advantage: we no longer need to record all k pointers. Instead, we just record the node that is closest. Thus, k becomes only a constant factor on the cost of JOIN. For any value of k, we maintain only $3 \lg n$ pointers per node. This is a unique advantage that SkipNet (and Family Trees) have over Pastry and Tapestry since the latter do not record name IDs.

In Family Trees, we have another simple way to trade of space for time: instead of just keeping pointers to two parents (one of each gender), we could keep pointers to ancestors in higher levels. For example, if we keep pointers to grandparents, then we can advance by 2 levels per hop, decreasing the number of hops per request by a factor of $\frac{1}{2}$. If we keep pointers to great grandparents, we can decrease the number of hops per request by a factor of $\frac{1}{3}$. However, the further we advance per hop, the more pointers we must maintain: if each pointer points to a node t levels above the current one, then there are 2^t different ancestor lists that we must point to. Thus, t must be a very small constant. (For nodes in high levels, these pointers may point to levels that are empty. In that case, we substitute pointers from lower levels.)

This is an optimization that Family Trees can use to take advantage of the fact that each Family Tree node requires fewer pointers for the basic routing protocol. Even in a network with only 1000 nodes, SkipNet will need $3 \lg n \approx 30$ pointers per node for basic routing, while Family Trees need only 9. If we allow both protocols to use 30 pointers per node, then Family Trees have 21 extra pointers to work with. For

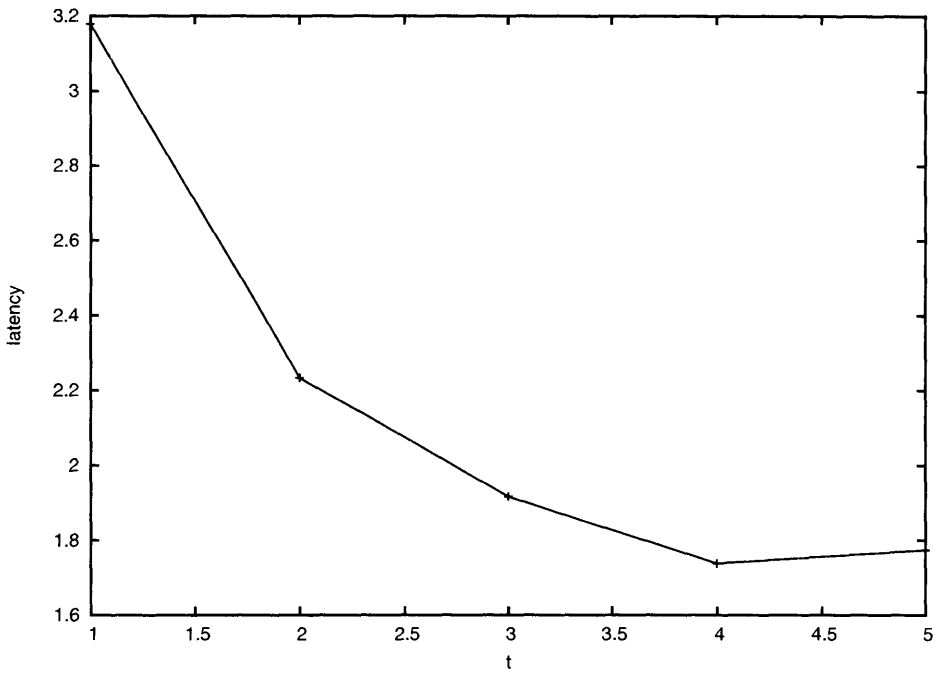

Figure 5-3: *Shows the relationship between t used in the Pastry heuristic and the latency of a lookup (normalized so that* 1 *is the average point-to-point latency of the network).*

the optimization just described, we could choose $t = 4$ (for 16 pointers), for example, and still have room to spare. This would allow the lookup algorithm to advance 4 levels per hop, which with 1000 nodes, means just 2.25 hops per lookup on average to reach the top level!

Figure 5-3 shows the improvement in latency achieved by increasing t. Again, we see diminishing returns. This is due to the fact that we need to take at least two hops in all cases, the latter of which will have very high latency. As we increase t, we can remove many of the other hops (which actually have much lower latency), but we cannot eliminate the last two.

We actually see slightly worse performance with $t = 5$ than $t = 4$. This is most likely due to the fact that most lookups require only 2 hops in both cases, but when $t = 5$, the first hop we make is to a node that is at a higher level and usually further away than when $t = 4$. In other words, we see improvement (with diminishing returns) as we increase t until we reach the point where we can traverse in 2 hops.

Then, increasing t actually increases latency. If we increased t to the point where we could advance to the top level in only 1 hop, then latency would decrease again.[5]

Now that we have seen how to eliminate the first part of numeric ID lookup (finding a level-0 node) and how to improve the performance of the second part (moving upward through the levels), all that remains is to look at the last part (linear search through the numeric ID list). This part is undoubtedly the worst performing in terms of latency. We know that we will take approximately $\lg n$ hops. But since these hops are between neighbors in the numeric ID list, they are essentially hops between random nodes in the network. In other words, the expected latency of each hop is equal to the average latency in the network. It is important to remember that the total latency of a lookup in SkipNet is only about 1.7 times the average latency in the network. Thus, even if we removed all latency in the first two parts of the algorithm, we could only afford to take 1.7 hops on average in the numeric ID list (instead of the $\lg n$ hops that are expected). In short, this part of the algorithm is simply incompatible with low latency. If we want Family Trees to have low latency, then we must eliminate it all together.

To describe how we will do this, we need a bit of terminology. We will refer to the set of nodes that believe they are at the top-most level, according to their own estimate of $\lg n$, as *top nodes*. At the end of the second part of a numeric ID lookup, we normally end up at a top node. The destination node will normally be between that top node and the next closest top nodes in the numeric ID list. In particular, we should end up at the top node that shares the most numeric ID bits with the destination. We will refer to that as the top node of the destination. In general, every node can assign itself a top node in this manner. Finally, we will refer to each set of nodes whose top node is the same, along with the top node itself, as a *cluster*.

Viewed this way, the purpose of the second part of the lookup algorithm is to find

[5]More generally, we see performance improvement when t is increased so that the number of hops needed to reach the top is decreased. However, when t is subsequently increased but before the next drop in number of hops, we will see see performance degradation due to being at higher levels for each hop. Since the number of levels is not fixed but random, we will see both effects occurring to some degree as t increases. It is only when the number of hops becomes very small (around 2) that the worsening effect outweighs the improvement effect.

the cluster containing the destination. And the purpose of the third part is to find the destination node within that cluster. Since the nodes of the cluster are randomly chosen, linear search is just about the worst algorithm we could possibly use. We could imagine other approaches like doing a second lookup over just these nodes. However, this would still require about $\lg \lg n$ high-latency hops, whereas we need to compete with SkipNet's 1.7. The approach we take is remove the restriction that the lookup must be routed to a particular node within the cluster. Instead, we will let all nodes of the cluster share responsibility for all keys (and their values) that are routed to nodes in that cluster.

There are many ways in which the nodes of a cluster could share responsibility for the keys. For example, we could make all nodes have copies of the keys, but only store the value (or the document) at the node in the cluster with closest numeric ID. The top node could forward each request received to the appropriate node within the cluster. This would require $\lg n$ pointers in the top node, which is less than the $3 \lg n$ pointers needed by a SkipNet node, but we would then no longer be degree optimal. Furthermore, this would still require another high latency hop, which would make it hard for us to compete with SkipNet.

The approach we take is to store all keys and values at every node in the cluster.[6] This approach has several advantages. First, it means that we don't (normally) need any more hops once we reach a cluster: in other words, the third part of the algorithm has been eliminated. Second, we can replace any parent pointer that points to a top node with a pointer to the node in that cluster that is closest to the node in question. This requires a $\lg n$ times as many requests in JOIN, but it will drastically reduce the latency of the last (and slowest) hop. Third, if the number of keys being stored is $\Omega(n)$ (which it almost always will be), then the expected number of keys at each node is $\Omega(n \lg n)$, which means that the number of keys at each node is within a constant factor of its expected value with high probability. This should eliminate the need to use virtual nodes [14] or more sophisticated techniques such as [15] to ensure load

[6]When a new key-value pair is inserted at a node, we can forward it around the cluster. If we require each insert to be atomic, then we would need to delay the response while the nodes within the cluster communicate (which will take $O(\lg n)$ hops).

balance. Fourth, because each key is stored at $\lg n$ nodes that are randomly chosen, the probability that they will all fail is negligible ($1/n^c$), so every key is assured to survive random node failures with high probability.

If an extra factor of $O(\lg n)$ on the space requirements is a problem, we could instead have nodes just maintain a fixed size cache and revert to linear search for cache misses. This would allow us to trade off between time and space. However, this does mean that we would need to resort to other techniques like virtual nodes to ensure load balance. And we will almost certainly need to maintain multiple copies of each key anyway to make sure it survives random node failures. In our study, we will assume that disk space will be sufficient such that every node can store all keys for the cluster.

Clustering also eases the implementation of Family Trees in a few ways. We link the top nodes into their own list sorted by numeric ID. Then the top nodes can look at their neighbors to know exactly how many bits are needed to distinguish its numeric ID from the those of other top nodes. It can use this as its choice of the number of levels. The expected number of levels, computed in this manner, will be roughly $\lg n - \lg \lg n$. The top node then sets its level to this value. Each node in the cluster chooses a random level between 0 and this value minus 1. In a normal Family Tree, each node chooses a level between 0 and $\lfloor \lg n \rfloor - 1$ (on average), which means that many of them end up in levels beyond $\lg n - \lg \lg n$, where they will most likely never be reached. This approach places more structure on the Family Tree, making it easier to ensure correct lookup algorithms as optimizations are added. Another effect of this is that each level has roughly $\lg n/(\lg n - \lg \lg n)$ times as many nodes as it would have in a normal Family Tree. This has the advantage that it makes empty level lists less likely, especially when n is small, which means Family Trees should perform more smoothly for small n. The disadvantage is that these nodes can increase the number of hops needed per lookup, as we will see in the section 5.3.

We performed one further optimization in our implementation. The second part of numeric ID lookup identifies a cluster whose numeric IDs match the maximum number of bits of the key. However, there may be multiple clusters that maximally match.

The expected number of matching clusters is a small constant. In our implementation, we chose a small increase in space over a small increase in latency: we allowed all of the maximally-matching clusters to share responsibility for those keys. We allowed SkipNet to use this optimization as well.[7] (However, the increase in latency that we would get if we forwarded to the cluster whose numeric IDs were closest modulo 1 would likely be fairly small using our clustering approach since we would be forwarding to the closest of the nodes in the other cluster.)

To implement this optimization, each cluster maintains a bit vector that records, for each i from 0 to the number of levels, whether the cluster is maximally-matching for a key that matches that many bits of its numeric ID. We can say "its numeric ID" because all of the nodes in the cluster should share the prefix of the top node's numeric ID of length equal to the number of levels.[8] It is important to note that this is $O(\lg n)$ *bits*, the size of one pointer, say, so this is only $O(1)$ words of space. This bit vector is important because sometimes a cluster will receive a request for key that should have been routed elsewhere. This often occurs when a level list on the path to the intended cluster was empty. In these cases, each cluster needs to know that it is not the intended destination, which it can determine by examining the bit vector. If it is not the intended destination, then it will forward the request to next cluster (in the numeric ID list) in the direction of the destination.

We have now seen all of the numeric ID lookup algorithm, optimized for latency. The last two optimizations described remove the linear search of the numeric ID list in every case except where pointers are missing from the level lists. Next, we will compare the latencies of lookups in Family Trees and SkipNet. We will first compare their performance without the optimization of routing to the closest node in the

[7]This optimization is one reason why our stretch values for SkipNet look better than what they reported.

[8]Because each cluster node falls between its top node and the next closest top node by numeric ID, it must share more bits with its top node than the other tope node. However, the number of levels chosen by the top node is the maximum of the number of matching bits with the top node before and the top node after. Thus, a cluster node may match more bits than the previous top node but less than the next top node. Normally, this will not occur. For the purposes of this algorithm, each cluster node will defer to the numeric ID of the top node if its bits differ over the relevant prefix.

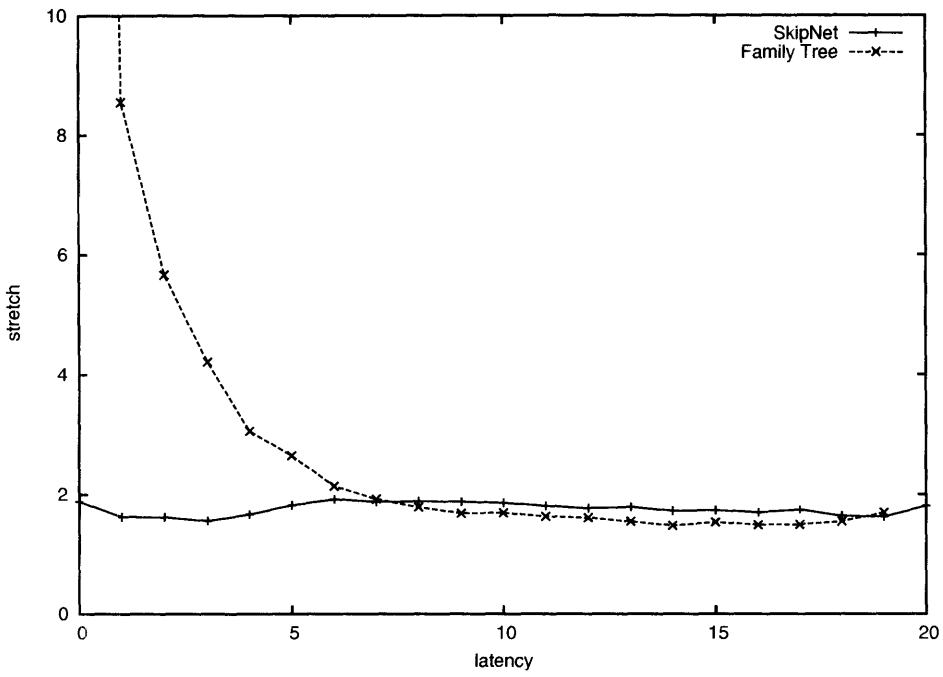

Figure 5-4: *Shows the stretch of Family Tree and SkipNet numeric ID lookups as a function of the latency between the start and destination nodes. The latencies are first normalized so that* 1 *is the averaged latency in the network and then multiplied by* 10.

cluster. We will look at that optimization subsequently. For the first comparison, we just route every request to the top node of the appropriate cluster.

Comparing the performance of this algorithm to that of SkipNet lookups is complicated by the fact that the performance of Family Trees is highly affected by the latency from the start to destination nodes. Figure 5-4 shows Family Trees with $t = 4$ versus SkipNet.[9] While SkipNet's performance seems to have little correlation with this latency, the performance of Family Trees is highly affected. To see why, note that each Family Tree lookup starts out by forwarding to the nearest level-0 node. If the destination is very close, the nearest level-0 node could actually be farther away. In addition, while the level-i lists in SkipNet have $n/2^i$ nodes (on average), the level-i lists in Family Trees have fewer, roughly $n/2^i \lg n$ nodes. Thus, for nodes that are separated by $O(\lg n)$ nodes in the name ID list, almost every hop between levels in a

[9]The numbers shown in the figure are from just one experiment with each algorithm. However, the trends shown were consistent across many runs.

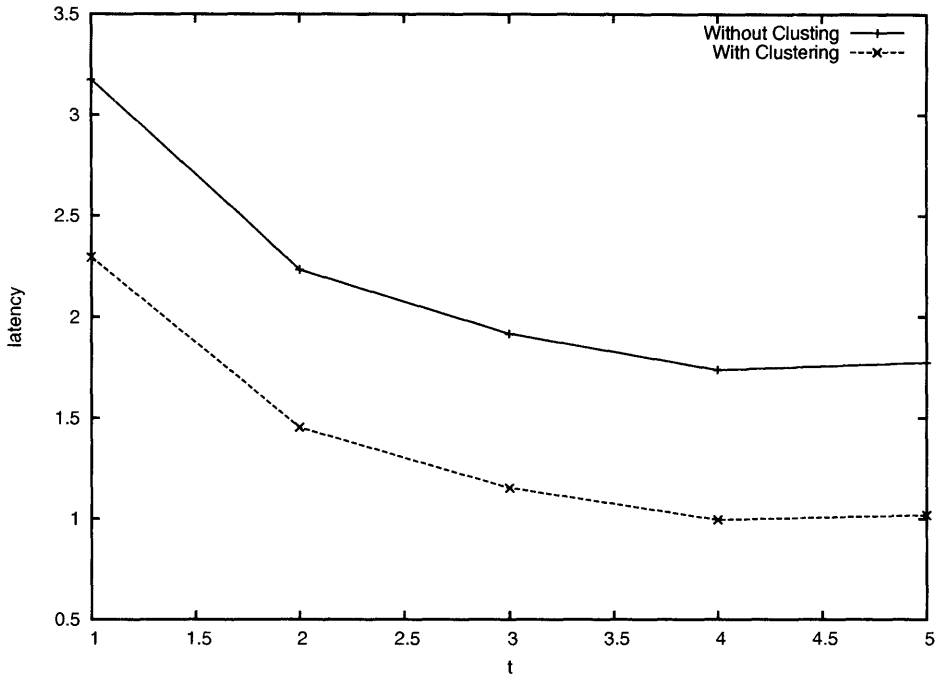

Figure 5-5: *Shows the latency of numeric ID lookups for Family Trees with and without clustering, as functions of t.*

Family Tree will be relatively large.

We can see in Figure 5-4 that SkipNet outperforms Family Trees for lookups where the latency between the start and destination less than 0.8 times the average in the network. When the latency between the start and destination is larger than this, Family Trees have the advantage. Their stretch in this range is about 1.6 compared to 1.7 for SkipNet. In practice, we are less concerned with the stretch of lookups when the latency between start and destination is small since these are already fast. So it seems fair to say that Family Trees are competitive with SkipNet.

Lastly, we look at performance when we route to the closest node in the cluster. It is no longer useful to look at stretch since we are changing the destination node. Instead, we will look directly at latency. Figure 5-5 shows the latencies of lookups, as functions of t, with and without clustering. We can see that, without clustering, the latency values are about the same as the stretch we saw earlier. The latency value shown is the query latency divided by the average latency in the network, while stretch is the query latency divided by the actual latency between the start and destination

nodes. Since the average value of the distance between the start and destination is the average value in the network, these values are very similar. Thus, clustering decreases the latency to about what we would see for a stretch of 1.0, a huge improvement over both SkipNet and Family Trees without clustering.

It is also possible for SkipNet to implement the clustering technique just described. In fact, SkipNet can implement an even stronger form. With only $O(1)$ pointers, a Family Tree node can emulate the top node of its cluster. This just takes two extra pointers: each node in the cluster needs pointers to the previous and next top nodes in the numeric ID list. If each node has these, then any pointer to the top node can be replaced with a pointer to any other node in the cluster. Unfortunately, with only $O(1)$ pointers, a Family Tree node cannot emulate the other $O(\lg n)$ nodes in the cluster. That would require $O(\lg n)$ pointers per node: each node would need to record the LEVELNEXT and LEVELPREV pointers of every other node in the cluster. In SkipNet, however, this emulation is possible.

In SkipNet, we can do the following. We form clusters of nodes as in Family Trees. The nodes are connected into a list sorted by numeric ID. Each estimates $\lg n$ (by looking at its successor's numeric ID) and chooses a random level to decide if it is a top node. Then, each node finds the neighboring top node whose numeric ID matches the most bits of its own. All nodes with the same top node (a cluster) should have the same prefix in their numeric IDs, which includes all of the bits that will be relevant for routing.[10] Next, we determine the LEVELNEXT and LEVELPREV pointers in the usual manner, except that we skip over nodes in the same cluster and we replace whatever node would then be found by the closest node in its cluster. As we can see, since all the nodes in the cluster are identical with respect to routing, we can optimize every pointer to point to the closest node in the cluster. In Family Trees, each node can only emulate the top node of its cluster, so we can only replace pointers to top nodes with pointers to the closest node in the cluster.

Figure 5-6 shows the sort of improvement we can expect from this type of clustering (full clustering) over the clustering of Family Trees (partial clustering). The

[10] If they do not match, the nodes defer to the top node's numeric ID bits.

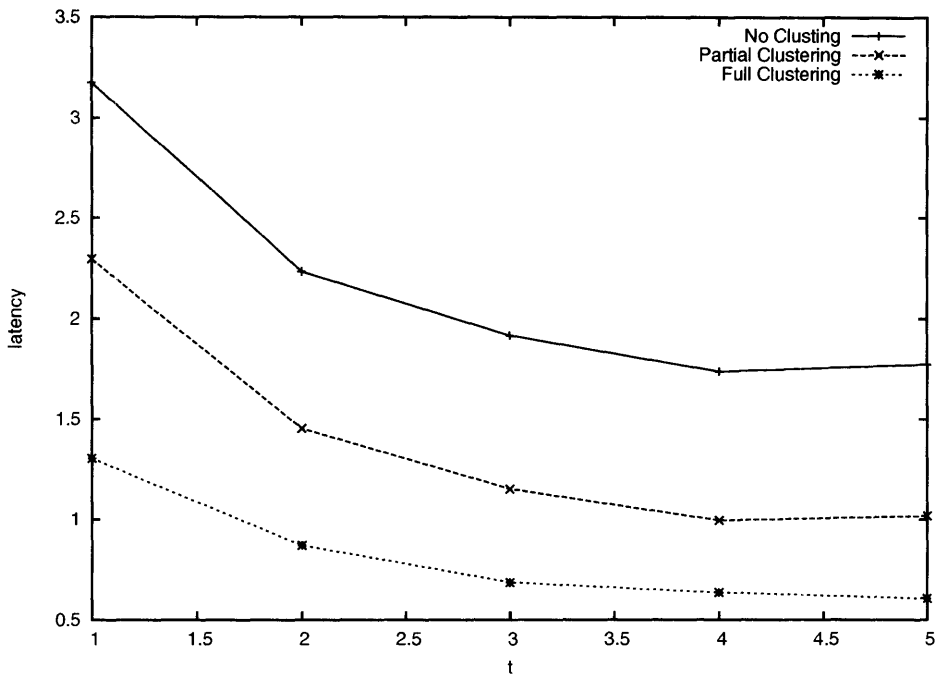

Figure 5-6: *Shows the latency of numeric ID lookups for Family Trees with no clustering, partial clustering, and full clustering, as functions of t.*

experiment was done by allowing Family Tree nodes to cheat and emulate other nodes of their cluster. Since Family Trees with $t = 4$ and SkipNet, both without clustering, have comparable performance, we should expect SkipNet to perform as well or better than what is shown here (for $t = 4$ or $t = 5$). In particular, SkipNet should achieve an average latency of 0.6 (or perhaps a little less) times the average latency in the network. This is quite an improvement over the normal 1.7.

In this section, we have seen how Family Trees can be made very efficient in terms of latency. In particular, they can be competitive with a latency-optimized SkipNet, even without clustering. However, applying the technique of clustering to SkipNet would give performance beyond what Family Trees can offer.

5.3 Verification of Theoretical Results

In this section, we look at how the size of the network affects performance. In particular, our goal is to verify the theoretical results of the previous chapter, which bounded

the performance in terms of hops per request and the congestion in the network by functions of the network size n. These bounds are important in practice as well: if hops per request or congestion is growing too quickly, then the algorithms will not scale to large network sizes. Even though good performance and congestion were proven, there are still some practical questions. For example, the hidden constants in the O-notation and high probability assumptions of the theoretical results could be hiding unreasonably large constants. The results of this section show that performance and congestion are good for small networks and grow slowly (as predicted) as the network size increases.

Our experiments will focus on lookups by name and numeric ID. As we saw in the previous chapter, the JOIN and LEAVE algorithms perform only a trivial amount of work except for a constant number of lookups. Thus, our experimental results showing good performance of the lookup algorithms immediately imply good performance for JOIN and LEAVE.

We first examine lookups by name ID. Figure 5-7 shows a plot of the average number of hops per lookup for networks of size $n = 100, 200, \ldots, 1000$. For each network of size n, we perform approximately 20 lookups per node in each trial and then average over 40 trials as described in section 5.1. For networks with less than 1000 nodes, say $100i$ nodes, we used only the first $96i/10$ stub nodes in the network so that the entire network is scaled down as n decreases (rather than just using less hosts over the same number of routers).

The dashed curve in Figure 5-7, which was fitted by linear regression, is the function $8.17(\lg n - 3.16 \lg \lg n + 3.58)$. The form of this function is complicated by several matters. First, some parts of the algorithm have performance proportional to $\lg n$ and others $\lg n - \lg \lg n$. Second, some parts have arguments that are not n but n/c for some constant c: the average distance between the nodes (in the name ID list) is only a fraction of n. But the most significant complication comes from the fact that we force all nodes in a cluster to choose levels between $\lg n - \lg \lg n$. The result is that each level list has $\lg n/(\lg n - \lg \lg n)$ times as many nodes as it would otherwise, which increases the number of sideways hops needed within each level list.

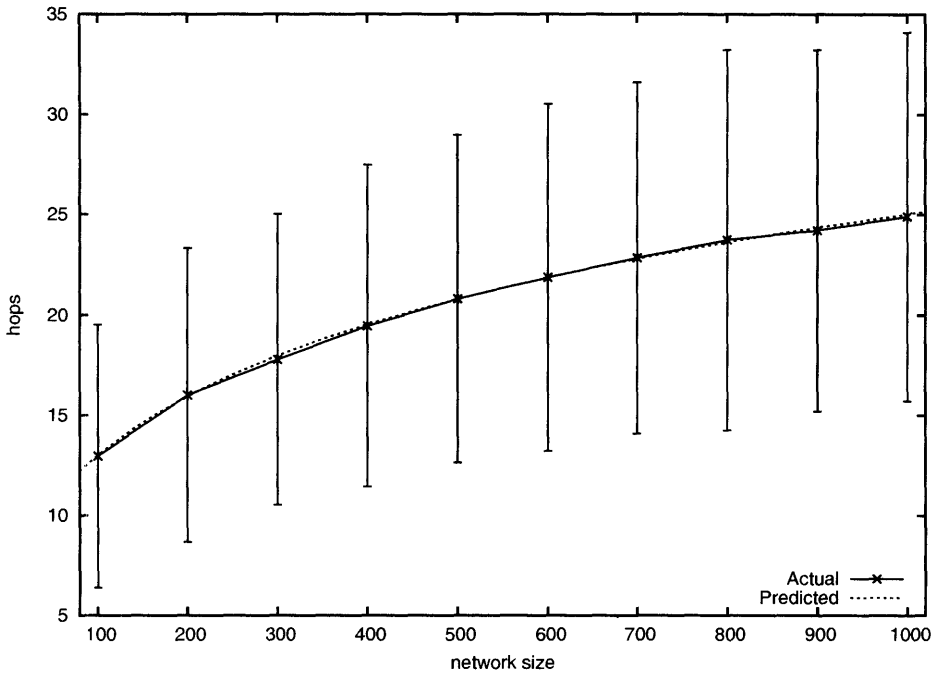

Figure 5-7: *Shows the actual and predicted number of hops per name ID lookup as a function of the network size n. The error bars on the actual numbers are one standard deviation.*

While this extra factor tends to 1, it adds lower order terms into the growth of the function.

Figure 5-7 shows that name ID lookups are demonstrating $O(\lg n)$ hops per request with a hidden factor of around 8. From the pseudocode of the last chapter, we might expect a constant factor closer to 5 or 6. The increase is again most likely due to the extra nodes in each level list. Nonetheless, 8 is not a particularly large constant. Growth on the order of $8 \lg n$ should not cause scalability problems for any reasonable values of n. Thus, these numbers confirm the practicality of Family Trees for performing name ID lookups.

Figure 5-7 also has error bars showing one standard deviation away from the mean. We can see in the figure that the confidence intervals increase in size as n gets larger. For each n, the 95% confidence interval extends $(1.83 \pm 0.1) \lg n$ from the mean. These results match well with the theory. We saw in the previous chapter that we could not guarantee that the number of hops would be within some fixed

amount of the expected value with good probability. Instead, we showed that, for any probability p, there was some constant factor c such that the number of hops would not be more than c times the expected value ($c \lg n$) with probability p. The experimental results show that the theoretical bound was not lacking: as n increases, the probability of being within a fixed distance from the expected value goes to 0. Indeed, the experimental results demonstrate the theoretical one: the probability of staying within a fixed constant factor stays roughly the same as n increases.

To see the congestion in the network, we look at the number of requests per node. We saw in the previous chapter that the probability of a random request going through a given node is $O(\lg n/n)$. If we perform m requests, then the expected number of requests at a given node will be $O(m \lg n/n)$. We will look at n/m times the number of requests at a given node, which should be $O(\lg n)$. In fact, the average value of this will be exactly the same as the average number of hops because both averages produce the same sum: one just sums over nodes and the other sums over requests.[11] Thus, a more informative metric about requests per node is its standard deviation, which is plotted in Figure 5-8. The (dashed) fit curve, also produced by linear regression, is the function $6.11(\lg n - 3.53 \lg \lg n + 4.48)$, about $\frac{3}{4}$ of the number of hops per lookup.

It is clear from the figure that the distribution of requests is not uniformly random. If, for each request, we simply picked 25 nodes at random, the standard deviation of the plotted value would be about 1.2. However, we should not expect the distribution to be uniform. For example, level-0 nodes will receive more than the average number of requests because $O(\lg n)$ nearby nodes will send every request through them first. On the other hand, nodes in the top-most levels will receive less than the average number of requests because lookups for nearby nodes will not need to go up that high.

These deviation, however, are substantially better than what we would see for a routing structure with poor congestion. For example, in a perfect binary tree with

[11] Actually, the former is about 1 more than the latter because there is always one more request than hop.

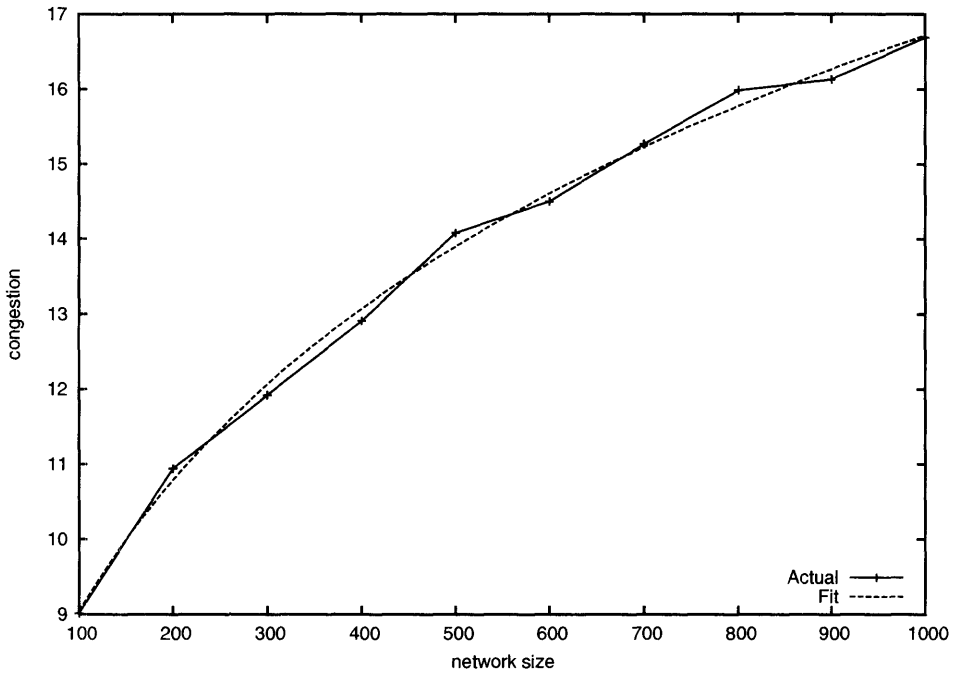

Figure 5-8: *Shows the standard deviation of the number of name ID requests per node, as a function of the network size n, along with the predicted values.*

1000 nodes, if we performed a number of requests so as to achieve the same average value[12], the standard deviation would be around 80, instead of 17 for the Family Tree. The improvement over a binary tree becomes even clearer if we look further out in the tails of the distribution. For example, both the binary tree and the Family Tree have 90% of the nodes with values less than 50 (about 2 times the average value). However, while the Family Tree has 95% of nodes less than 55, the 95-th percentile for the binary tree is at 100. The 99-th percentile for Family Trees and binary trees are around 65 and 375, respectively. And finally, the maximum value for Family Trees and binary trees are around 100 and 1000, respectively. Thus, Family Trees show substantial improvement in congestion over binary trees. Similar results would be seen for Skip Lists and other traditional data structures.

Next, we turn to numeric ID lookups. Figure 5-9 shows the same plot as in Figure 5-7 but for numeric ID lookups. This time the dashed curve is the simpler function $\lg n - \lg\lg n + 1$. This is number of hops we should expect: it is the expected

[12]The average lookup in a binary tree would perform about 3/5 as many requests as in a Family Tree, so we must perform 5/3 as many lookups to get comparable numbers.

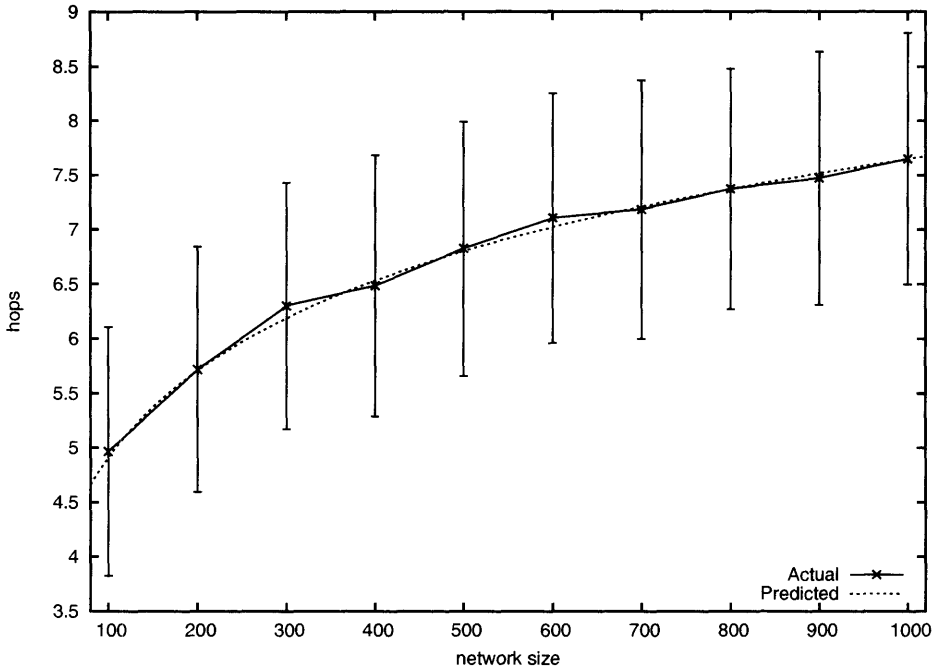

Figure 5-9: *Shows the actual and predicted number of hops per numeric ID lookup as a function of the network size n. The error bars on the actual numbers are one standard deviation.*

number of levels needed to get to an empty list plus the one hop to a nearby level-0 node.[13] As we saw in section 5.2, we are skipping the linear search through the numeric ID list that was described in the algorithm of the previous chapter. Thus, the only significant work is moving up through the levels. Also note, that unlike name ID lookups, we do not get any benefits of locality: every lookup must go all the way to the top-most level. This is why we do not have a negative additive constant.

The error bars in Figure 5-9 differ from those in Figure 5-7 in that they do not appear to be growing with n. This can also be understood from the theory. While the linear searches in the name and numeric ID lists were only shown to take $O(\lg^2 n)$ hops with high probability, the upward movement in numeric ID lookup takes as many hops as levels, and there are $O(\lg n)$ levels with high probability. For this reason, we should expect the number of requests per numeric ID lookup, now that we have eliminated the two linear searches, to be more tightly clustered around the

[13]The number of hops should be one less than the number of levels, but the implementation artificially chooses the number of levels one larger than needed, which cancels that out.

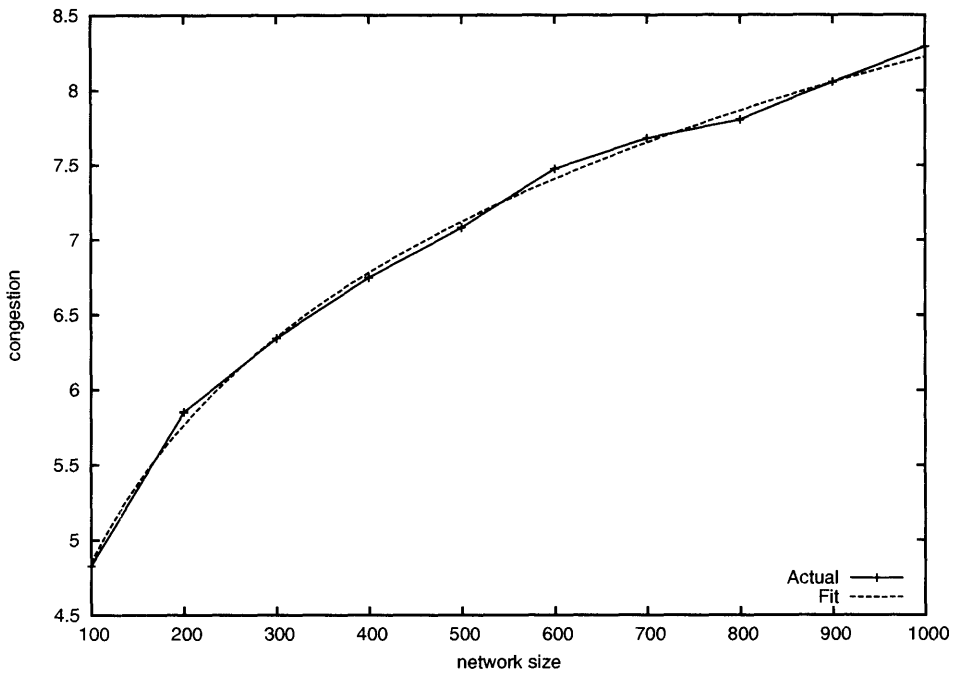

Figure 5-10: *Shows the standard deviation of the number of numeric ID requests per node, as a function of the network size n, for two independent experiments, along with the predicted values.*

mean than those of name ID lookups, which still include a linear search in the name ID list.

The results for congestion during numeric ID lookups, shown in Figure 5-10, are similar to those for name ID lookups. The dashed fit curve is the function $1.66 \lg n - 3.65 \lg \lg n + 3.79$, which is only slightly larger than the average number of hops per lookup and still a slowly growing function of n.

Thus, we have seen that the number of hops per lookup and the congestion in the network are slowly growing functions of n as predicted by the theory. Futhermore, the algorithms perform well even for small values of n. And since the hidden constants are reasonably small, we should not see scalability problems in practice.

5.4 Name-Constrained Lookups

As we saw in the second chapter, if we take a range of name IDs $[a, b]$ and pretend that all SkipNet nodes whose names aren't in this range disappeared, the remaining nodes form a completely normal SkipNet. This means that we can perform a numeric ID lookup constrained to nodes over just that range. As we saw in chapter three, the name-constrained numeric ID lookup is very useful in practice. Furthermore, any time a sub-domain becomes disconnected from the network, we are essentially performing name-constrained lookups (as long as the names used are domain names), so the ability to perform such lookups efficiently shows that SkipNet can efficiently survive a sub-domain disconnect.

Unfortunately, Family Trees do not share this property. If we remove all nodes whose name IDs are not in the range $[a, b]$, the Family Tree formed by the remaining nodes is "incorrect" in two respects. First, each node estimates its level incorrectly. In particular, the estimates are too large, which means that the levels are sparser than they should be, and the odds of finding an empty level (which prevents further routing) are increased. Second, the numeric ID list is now disconnected. This means that, if we encounter an empty level and do not reach the intended cluster during the upward part of a numeric ID search, we have no way of finding the right node.[14]

We can fix these problems in the special case in the case where the name IDs are hierarchical (with only $O(1)$ parts) and we are only concerned with constraining the names to a particular sub-domain (i.e., to nodes with names of the form $c_1.c_2.\cdots.c_k.*$, where the c_is are fixed values). In our experiments, we used domain names, which are hierarchical and have at most 4 or 5 parts in practice. These are the most important type of name IDs, so this will allow Family Trees to perform name-constrained numeric ID lookups for the most important cases in practice.

To fix the second problem (a disconnected numeric ID list), we create an additional numeric ID list for each prefix of name IDs. For name IDs with four parts, the nodes will participate in 4 numeric ID lists. The first list is the normal (unconstrained)

[14]We could enumerate the nodes in the name ID list and check *every node's* numeric ID, but this would be hopelessly inefficient

numeric ID list. Next, is the sublist containing only nodes with name IDs of the form $c_1.*$. Similarly, we have a list for $c_1.c_2.*$ and $c_1.c_2.c_3.*$. These additional lists require 6 extra pointers per node.

To get an acceptable theoretical solution to the first problem (levels being too sparse), we could insert each node into multiple different Family Trees, one for each prefix of its name ID. This would require 15 extra pointers per node, in addition to those for the constrained numeric ID lists. It is not clear, however, that sparse levels are a real problem in practice. For example, suppose that a sub-domain containing $\sqrt{n}$ nodes were disconnected. (If $n = 1000$, the sub-domain contains only 32 nodes.) In the resulting network, each level list would have a length of roughly $\sqrt{n}/2^i \lg n$ instead of $\sqrt{n}/2^i \lg \sqrt{n}$. But since $\lg \sqrt{n} = \frac{1}{2} \lg n$, each level list is only half as full as it should be. As a result, we expect to reach an empty list only 1 level before we would normally. This increases the expected number of nodes searched in the numeric ID list by only a factor of 2. For an arbitrarily small sub-domain, the linear search can be made arbitrarily large relative to the size of the sub-domain. However, for such small domains, even a linear search will be fast since there are so few nodes. Theoretically, we can get bad performance in some cases. For example, if the sub-domain contains $\lg^k n$ nodes, then we will end up searching $\lg^{k-1} n$ nodes, which is not particularly efficient for large k. Nonetheless, it is not clear that this will translate into poor performance in practice. Thus, in our implementation, we simply ignored the problem of sparse levels and let experiment be the final judge.

One final problem is that we cannot use our normal clusters for name-constrained searches since these are based on the unconstrained numeric ID list. However, we can make separate clusters for each constrained numeric ID list. This only requires 2 pointers for each name ID prefix.[15] This optimization was not implemented in our experiments, however, so the results show "full price" for these lookups.

Figure 5-11 shows the average number of hops per request, with error bars at one standard deviation. The x axis is the number of parts of the name ID that were

[15]This can be reduced to just a single pointer if we are willing to forward every request that does not reach the correct destination to the top node of the cluster before it can be forwarded to the next cluster.

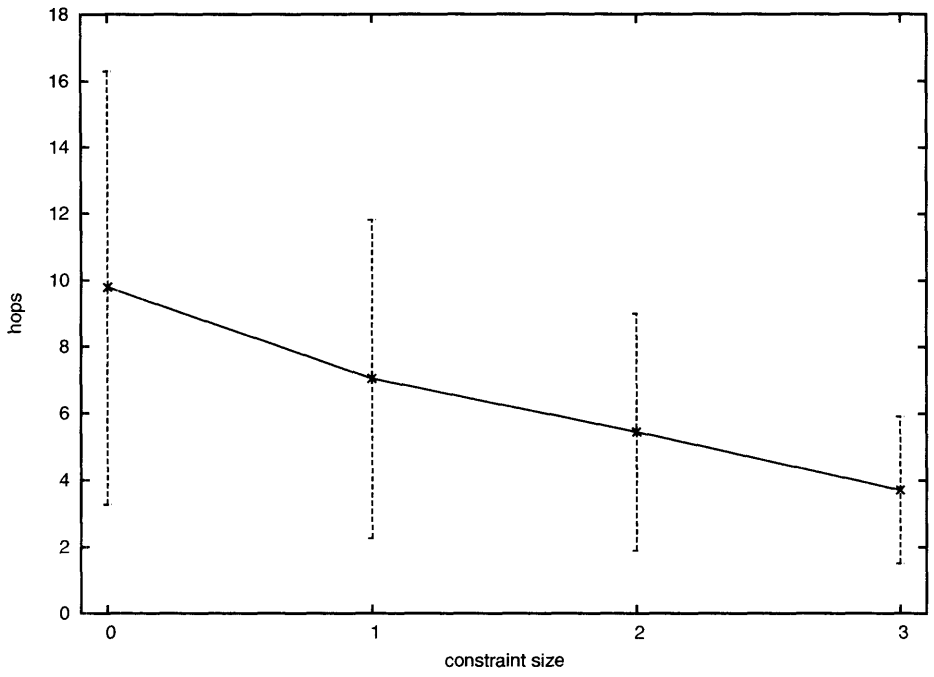

Figure 5-11: *Shows the average number of hops per request, with error bars at one standard deviation, for numeric ID lookups with various amounts of the name ID constrained.*

constrained in the search. So $x = 0$ is a normal numeric ID lookup. (Recall that clustering and shared responsibility between clusters are turned off in this experiment.) Figure 5-12 shows the average latency of these requests. In both figures, we see that as the constrained sub-domain gets smaller, the performance of the lookup improves. In all cases, the number of hops and latency look quite reasonable. Thus, these results show that Family Trees can perform name-constrained numeric ID lookups efficiently in the most important practical cases at the expense of $O(1)$ more pointers per node.

As in SkipNet, these name-constrained numeric ID lookups will be unaffected by a sub-domain disconnect as long as the constraint is completely inside or outside of the disconnected sub-domain. These results also imply that name ID lookups within a disconnected domain should perform well. Such lookups would only suffer from the problem of having levels that are too sparse. But as we saw above, this theoretical problem does not seem to translate into poor performance in practice. Thus, we can see that Family Trees can also perform better than unordered networks when a

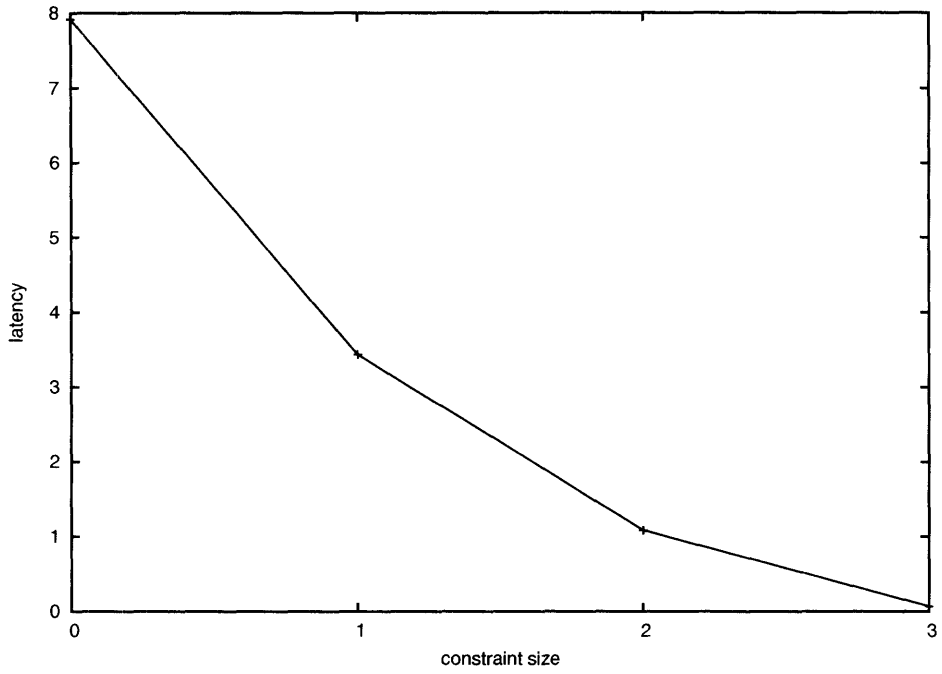

Figure 5-12: *Shows the average latency per request for numeric ID lookups with various amounts of the name ID constrained.*

sub-domain becomes disconnected from the network.

5.5 Conclusions

In this chapter, we have seen that Family Trees are a practical peer-to-peer overlay network. We have shown that their performance in practice matches the theoretical expectations fairly well. The hidden constants in the theory are not unreasonably large. We have also seen that Family Trees can be optimized very aggressively to achieve performance in terms of latency that is competitive with the SkipNet, Pastry, and Tapestry (although we also discovered that SkipNet could also be optimized to improve its performance even further). Lastly, we have seen that Family Trees can perform name-constrained numeric ID lookups in the most important practical cases at the expense of $O(1)$ more pointers per node. This ability greatly increases their usefulness in practice.

Chapter 6

Conclusions

Peer-to-peer overlay networks are an important tool for organizing nodes in a distributed system. They are highly available because they lack single points of failure. They distribute the load of requests and network traffic evenly across the system so that bottlenecks do not form. They can provide distributed storage over the nodes of the network. And they can efficiently multicast to nodes in the network. These features are extremely useful for building distributed systems. Indeed, peer-to-peer overlay networks could become a standard service provided by every distributed applications platform.

Ordered networks were an important advancement in peer-to-peer systems. They allow lookups using "friendly names" rather than IP addresses (without resorting to a less scalable, external system like DNS). They often can gracefully survive an administrative domain being disconnected from the rest of the network. They can restrict the storage of documents or the network traffic for a given request to a particular administrative domain. And they can efficiently find all nodes within a particular administrative domain. These features greatly enhance the manageability of the peer-to-peer network, which increases the likelihood that such systems will see major deployments in practice.

These features also translate into additional operations available to applications, including range queries and name-constrained lookups. These operations extend the set of applications that are easy to build on top of the system. We saw in chapter three

that ordered networks can easily be used to implement a mobile entity location system. Furthermore, we saw how SkipNet, the first ordered network with efficient lookup performance, can be optimized to provide optimal efficiency for such a system. This application underscores the usefulness of ordered networks for developing applications.

Degree-optimal networks were another important advancement in peer-to-peer systems. Such networks use only a constant number of "pointers" per node to implement routing. Hence, these networks can be implemented on systems with only a very small amount of available resources. When more resources are available, they can be used to further improve the performance of the system. Or the additional resources can be left for use by the application, which undoubtedly has its own need for memory and network resources.

Thus, ordered and degree-optimal networks provide features that are useful to applications. This thesis presented Family Trees, the first peer-to-peer overlay network that is both ordered and degree-optimal. Prior to this work, it was unknown whether both properties could be achieved by the same system.

In chapter four, we presented the theory of Family Trees. We defined their structure and the algorithms for implementing their operations. We proved the correctness of these operations. We also proved that Family Trees are efficient: all of their operations require $O(\lg n)$ hops in expectation and $O(\lg^2 n)$ hops with high probability. Furthermore, we proved that all of these operations maintain optimally low congestion in the network.

In chapter five, we discussed Family Trees in practice. We described our own implementation of Family Trees on top of the `p2psim` network simulator. Our implementation included several optimizations to improve practical performance in terms of latency. We showed the results of experiments demonstrating that Family Trees can achieve very good practical performance. Family Trees perform suitably when few resources are available, but as more resources are provided, they can be used to further improve the performance of the system. Even in a network of 1000 nodes, Family Trees are competitive with existing SkipNet implementations when both systems are given equal resources, but Family Trees can continue to operate when less

resources are available. (We also described how some of our optimizations could be applied to SkipNet to improve their performance even further.) We showed the results of experiments verifying the theoretical bounds on performance in terms of the size of the network. These demonstrated that the network and CPU utilization as well as congestion grow slowly with the size of the network. Lastly, we described how Family Trees can implement name-constrained lookups in the most important practical cases, and we showed experimental results demonstrating their good performance.

This thesis has provided an important advancement to the study of peer-to-peer networks, with the development of Family Trees. They are an advancement to the theory of peer-to-peer networks as the first system that is both ordered and degree-optimal. They are also a practical and efficient system, suitable for use in real-world applications.

Bibliography

[1] James Aspnes and Gauri Shah. Skip Graphs. In *14th ACM-SIAM Symposium on Discrete Algorithms*, January 2003.

[2] Salman A. Baset and Henning Schulzrinne. An analysis of the skype peer-too-peer internet telephony protocol, 2004. http://arxiv.org/abs/cs/0412017.

[3] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. In *Microsoft Technical Report #MSR-TR-2002-82*, 2002. http://www.research.microsoft.com/~antr/PAST/location.pdf.

[4] Yan Chen, Randy H. Katz, and John Kubiatowicz. Dynamic replica placement for scalable content delivery. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, March 2002.

[5] Russ Cox, Athicha Muthitacharoen, and Robert T. Morris. Diminished chord: a protocol for heterogeneous subgroup formation in peer-to-peer networks. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems*, February 2004.

[6] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[7] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 448–455, 2003.

[8] Anh-Tuan Gai and Laurent Viennot. Broose: A Practical Distributed Hashtable based on the De-Bruijn Topology. In *Proceedings of the Fourth IEEE International Conference on Peer-to-Peer Computing*, August 2004.

[9] Thomer M. Gil, Frans Kaashoek, Jinyang Li, Robert Morris, and Jeremy Stribling. p2psim: a simulator for peer-to-peer protocols, 2004. http://www.pdos.lcs.mit.edu/p2psim/.

[10] Steven D. Gribble, Eric A. Brewer, Joseph H. Hellerstein, and David Culler. Scalable, distributed data dtructures for internet service construction. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation*, October 2000.

[11] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: a tool to estimate latency between any two internet hosts, from any internet host. In *Proceedings of ACM SIGCOMM*, November 2002.

[12] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *USENIX Symposium on Internet Technologies and Systems*, March 2003.

[13] M. Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[14] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[15] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 36–43, 2004.

[16] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufman, 1992.

[17] W. Litwin, M. A. Neimat, and D. A. Schneider. Lh* - a scalable distributed data structure. *ACM Trans. on Database Systems*, 21(4), 1996.

[18] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing.* ACM, July 2002.

[19] P. Mockapetris. Domain names – concepts and facilities, 1987. http://www.ietf.org/rfc/rfc1034.txt.

[20] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Workshop on Algorithms and Data Structures (WADS)*, 1989.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, August 2001.

[22] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[23] 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 ACM SIGCOMM*, August 2001.

[24] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Algorithms.* Prentice Hall, Upper Saddle River, New Jersey, 2002.

[25] Hongsuda Tangmunarunkit, Ramesh Govindan, Scott Shenker, and Debora Estrin. The impact of routing policy on internet paths. In *Proceedings of IEEE INFOCOM 2001*, April 2001.

[26] Udi Weider and Moni Naor. A simple fault tolerant distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[27] Kevin C. Zatloukal and Nicholas J. A. Harvey. Family trees: an ordered dictionary with optimal congestion, locality, degree, and search time. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2004.

[28] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM 1996*, April 1996.

[29] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.